Décidabilité et Complexité

Olivier Bournez, Gilles Dowek, Rémi Gilleron, Serge Grigorieff, Jean-Yves Marion, Simon Perdrix, Sophie Tison

Chapitre 1

Décidabilité et Complexité

1.1 Introduction à l'informatique fondamentale

L'informatique fondamentale est un vaste sujet, comme en témoignent les 2 283 et 3 176 pages des "Handbooks" (228; 1). Couvrir en quelques dizaines de pages, l'ensemble de l'informatique nous a semblé une entreprise hors de notre portée. De ce fait, nous nous sommes concentrés sur la notion de calcul, sujet qui reflète le goût et la passion des auteurs de ce chapitre. La notion de calcul est omniprésente et aussi ancienne que les mathématiques.

- Calculs sur les nombres entiers : algorithme d'Euclide calculant plus grand commun diviseur de deux nombres entiers, crible d'Eratosthène donnant les nombres premiers, etc.
- Calculs sur les nombres réels : méthode de Simpson de calcul numérique d'une intégrale, méthode de Newton d'approximation d'un zéro d'une fonction sur les réels, etc.
- Calculs géométriques : aires et volumes divers, algorithme de Fortune donnant le diagramme de Voronoï d'un ensemble de n points $A_1, \ldots, A_n$ du plan (ce diagramme décompose le plan en n régions d'influence de formes polygonales : la i-ème région est formée des points du plan plus proches de A_i que des autres points A_j, $j \neq i$), etc.
- Tri d'une famille d'objets selon un certain ordre : tris par insertion, échange (tri bulle, quicksort), sélection (tri par tas), fusion, distribution. Knuth (134) y consacre 388 pages !
- Calculs sur les textes : recherche de motifs (algorithme de Knuth-Morris & Pratt), compression de texte (algorithme de Ziv & Lempell).
- Calculs dans les graphes finis : plus court chemin entre deux sommets (algorithme de Dijkstra), couverture minimale (i.e. ensemble C de sommets tels que tout sommet soit dans C ou voisin direct d'un point de C).

0. Olivier Bournez, École polytechnique, bournez@lix.polytechnique.fr, Gilles Dowek, École polytechnique, gilles.dowek@polytechnique.edu, Rémi Gilleron, Laboratoire d'Informatique Fondamentale de Lille (LIFL), gilleron@univ-lille3.fr, Serge Grigorieff (Coordinateur), LIAFA, CNRS & Université Paris Diderot, seg@liafa.jussieu.fr, Jean-Yves Marion (Coordinateur), Nancy-Université, LORIA Jean-Yves.Marion@loria.fr Simon Perdrix, PPS, CNRS & Université Paris Diderot, simon.perdrix@pps.jussieu.fr. Sophie Tison, Laboratoire d'Informatique Fondamentale de Lille (LIFL), sophie.tison@lifl.fr

Ce n'est qu'au tournant du 20ième siècle que la notion de modèle de calcul a été définie et étudiée. Nous présenterons de nombreux modèles de calcul au cours de ce chapitre. La variété de ces modèles montre la difficulté à appréhender la nature et la diversité des calculs. Il y a les calculs mécaniques et séquentiels, les calculs quantiques, et tous les autres calculs qui s'appuient sur des hypothèses physiques de modélisation de la nature. Un modèle de calcul est une représentation mathématique de ce qui est calculable suivant certaines hypothèses sur le monde physique qui nous entoure. Ainsi face à face, il y a des modèles de calcul et des réalisations concrètes (les ordinateurs ...). Cette dualité réside aussi dans le rapport entre syntaxe et sémantique, entre l'objet manipulé et son sens. Notre première étape nous conduira à explorer différents modèles qui calculent les mêmes fonctions que les machines de Turing. Ceci étant, chacun de ces modèles propose une vision particulière qui tisse une notion de calcul. Il y a les calculs séquentiels et le développement de la calculabilité. La théorie de la calculabilité définit une représentation abstraite et algébrique de ce qui est calculable. Un de ces objectifs est ainsi de montrer ce qui est calculable/décidable de ce qui ne l'est pas, c'est à dire qui est indécidable. (à suivre)

1.2 Emergence de la notion de calculabilité

Si les notions intuitives d'algorithme et de calculabilité remontent à l'Antiquité, leurs formalisations mathématiques n'ont été réalisées que dans les années 1930 pour la calculabilité et seulement dans les années 1980 pour la notion d'algorithme, cf. §1.4.

1.2.1 Modèles de calcul discrets par schémas de programmes

Récursivité primitive. Les premières tentatives de formalisation mathématique de la notion de calculabilité se sont faites en considérant divers procédés de constructions de fonctions à partir d'autres fonctions. En termes actuels de programmation, ces constructions correspondent à des instructions bien connues. C'est ainsi que la première approche mathématique a été celle des fonctions appelées aujourd'hui "récursives primitives". Sur les entiers, il s'agit des fonctions que l'on peut obtenir à partir de fonctions très simples (fonction constante de valeur 0, fonction successeur, fonctions projections $\mathbb{N}^k \to \mathbb{N}$) par composition et définition par récurrence. En termes de programmation impérative, ce sont les fonctions programmables avec la seule boucle `FOR` (donc sans boucle `WHILE`). Cependant, on s'est assez vite aperçu (Ackermann, 1928) que cette famille était trop petite.

Systèmes d'équations de Herbrand-Gödel. La première bonne formalisation a été imaginée par Jacques Herbrand en 1931, quelques semaines avant sa mort (à 23 ans) dans un accident de montagne. Ayant, fort heureusement, décrit à grands traits ses idées dans une lettre à Gödel, c'est ce dernier qui en a assuré la mise en oeuvre. Cette formalisation, appelée aujourd'hui "systèmes d'équations de Herbrand-Gödel", permet des définitions à l'aide de n'importe quel système d'équations fonctionnelles. En termes actuels, on pourrait parler de programmation fonctionnelle.

Fonctions récursives de Kleene. C'est Stephen Cole Kleene, 1936 (132), qui introduit la formalisation moderne la plus utilisée, dite classe des fonctions récursives (ou calculables), à l'aide d'une nouvelle construction : la minimisation. Cette construction permet de définir, à

partir d'une fonction totale $f : \mathbb{N}^{k+1} \to \mathbb{N}$, une fonction partielle $g : \mathbb{N}^k \to \mathbb{N}$ telle que

$$\textit{domain}(g) = \{\vec{x} \mid \exists y\ f(\vec{x}, y) = 0\} \qquad g(\vec{x}) = \textit{le plus petit } y \textit{ tel que } f(\vec{x}, y) = 0$$

Sans surprise, on montre que la classe des fonctions récursives à la Kleene coïncide avec celle des fonctions programmables avec les deux boucles `FOR` et `WHILE`. Cette classe coïncide aussi avec celle des fonctions Herbrand-Gödel calculables.

1.2.2 Modèles de calcul discrets par machines séquentielles

Machines de Turing. Quelle que soit la force des définitions de Herbrand-Gödel et de Kleene, elles ne permettent pas de répondre avec conviction à la question "*peut-on avoir d'autres fonctions calculables ?*". C'est seulement avec l'analyse faite par Alan Mathison Turing, 1936 (224), que cette question peut enfin recevoir une réponse négative crédible. Dans cet article, Turing argumente de façon très convaincante sur le fait que ce qui peut être calculé par un humain qui travaille mécaniquement avec un papier et un crayon en un nombre fini d'étapes est calculable par le dispositif qu'il a imaginé, et est appelé maintenant "machine de Turing". Dans sa plus simple expression, ce dispositif est un ruban type machine à écrire sur lequel une tête de lecture/écriture peut lire et écrire des lettres d'un alphabet fini fixé et peut aussi se déplacer d'un cran à la fois dans les deux sens. L'écriture et le mouvement sont régis par l'état de la machine qui varie dans un ensemble fini fixé. On montre (c'est pénible mais sans réelle difficulté) que les machines de Turing (modulo des conventions d'arrêt selon certains états) permettent de calculer exactement les fonctions calculables à la Kleene.

Machines de Turing avec plusieurs rubans de dimensions quelconques et/ou plusieurs têtes. Une extension naturelle de la machine de Turing est de remplacer l'unique ruban par plusieurs rubans. Ou de multiplier les têtes. Ou de considérer des rubans de dimensions 2 ou 3 ou plus (avec des têtes pouvant se déplacer d'une case dans chaque direction). On montre que ceci ne permet pas de définir de nouvelle fonctions.

Machines de Kolmogorov-Uspenski. Afin de tester la robustesse de la définition de Turing, Kolmogorov et Uspenski, 1958 (137), imaginent de remplacer le ruban linéaire de Turing par un ruban en forme de graphe non orienté qui peut se modifier au cours du calcul. Lecture, écriture et modification du ruban (suppression ou adjonction d'un noeud) se faisant toujours de façon locale, là où la tête se trouve. On suppose que le graphe reste de degré borné fixé, c'est-à-dire que le nombre de voisins d'un sommet quelconque est toujours $\leq k$ pour un k fixé indépendant du sommet.

Machines de Schönhage (Storage modification machines). Allant encore plus loin, Schönhage, 1980 (195), imagine un ruban en forme de *graphe orienté* déformable dont le degré sortant est borné mais pas le degré entrant. L'intuition est la suivante. Les arcs du graphe sont vus comme des pointeurs : d'un sommet ne peuvent partir qu'un nombre $\leq k$ de pointeurs, mais un nombre arbitraire de pointeurs peuvent y arriver.
Mais là encore, machines de Kolmogorov-Uspenski et machines de Schönhage ne donnent rien de plus que les fonctions Turing calculables.

1.2.3 Modèles de calcul discrets par machines à registres à accès aléatoire (Random Access Machines)

Pour mimer le fonctionnement d'un ordinateur très idéalisé, Melzak, 1961 (164), et Minsky, 1961 (165), imaginent une machine avec une infinité de registres, chacun capable de contenir un entier arbitrairement grand. Elgot & Robinson, 1964 (81), montrent (voir aussi Cook & Reckhow, 1973 (52)) qu'en permettant l'accès indirect, c'est-à-dire la possibilité d'accéder à un registre à partir de son numéro lu dans un autre registre, et des opérations très élémentaires sur les contenus des registres, ce modèle (appelé *Random Access Machine* en anglais, c'est-à-dire *Machine à Accès Aléatoire*) calcule exactement les mêmes fonctions que les machines de Turing.

1.2.4 Grammaires de type 0 de Chomsky

Un autre modèle de calcul discret opère par manipulation de mots, ce sont les grammaires de type 0 de Chomsky. Il s'agit d'ensembles de règles de la forme $P\alpha Q \to P\beta Q$ (où P, Q désignent des contextes gauche et droits quelconques et α, β sont des mots fixés) qui permettent de remplacer un facteur α dans un mot par le facteur β. Partant d'un mot initial, on peut alors dériver toute une famille de mots. Avec quelques conventions simples, on peut considérer les fonctions dont le graphe est ainsi engendrable. On montre que les fonctions ainsi calculables sont encore celles qui sont Turing calculables.

1.2.5 Modèle de calcul discret à parallélisme massif : les automates cellulaires

Le modèle décrit ci-dessous ne permet pas non plus d'aller au-delà des fonctions calculables par machines de Turing. En revanche, son efficacité en temps est de beaucoup plus grande. De plus, le modèle est en soi fascinant : nous invitons le lecteur à taper "Conway Life" sur Google videos pour visualiser l'incroyable variété des comportements de cet automate cellulaire (dû à John Horton Conway, 1970) à deux dimensions qui n'a que deux états.

Avec la capacité technologique actuelle d'architectures comportant des millions de processeurs, un des challenges de l'informatique théorique aujourd'hui est de créer un nouveau paradigme de programmation pour utiliser pleinement un tel parallélisme massif.
Pour cela, il faut d'abord choisir un type d'architecture pour le parallélisme massif. Deux questions clés : la géométrie du réseau de processeurs et le fonctionnement synchrone ou non. Visionnaire, ici comme dans tant d'autres sujets, John von Neumann a mis en place dès 1949 un cadre théorique pour le parallélisme massif : celui des automates cellulaires. Et il a fait deux choix fondamentaux : *fonctionnement synchrone et géométrie simple à deux ou trois dimensions.* Chacun de ces deux choix a été depuis fortement questionné. D'autres approches ont été développées. Aucune n'a donné de nouveau paradigme de programmation pour le parallélisme massif. Si von Neumann visait des modèles de dimension 3 imitant certains mécanismes de l'activité cérébrale, (cf. (231) ou M. Delorme (65)), l'étude des automates cellulaires s'avère déjà fort complexe dès la dimension 1.

Formellement, en dimension 1, un réseau d'automates cellulaires indexé par $\mathbb{Z}$ (ensemble de tous les entiers) est la donnée d'un triplet (Q, δ, I_0) où Q est l'ensemble fini des états,

$\delta : Q^3 \to Q$ est la fonction de transition et $I_0 : \mathbb{Z} \to Q$ est la répartition initiale des états. Notant $I_t(i)$ l'état de la cellule en position i à l'instant t, l'évolution du réseau obéit à la loi suivante : $I_{t+1}(i) = \delta(I_t(i-1), I_t(i), I_t(i+1))$, i.e. le prochain état d'une cellule ne dépend que de son état actuel et des états actuels de ses deux voisines. La notion s'étend à des indexations par $\mathbb{N}$ ou $\{1, \ldots, n\}$ en rajoutant à δ des variantes pour le cas des bords (cas $i = 0$ pour $\mathbb{N}$ ou $i = 1, n$ pour $\{1, \ldots, n\}$). Elle s'étend aussi à des dimensions supérieures et, plus généralement, à des indexations par les sommets d'un graphe de degré d borné (d étant le nombre maximum de voisins d'un sommet).
Le choix de von Neumann du caractère synchrone apparaît comme un axiome puissant pour un développement théorique. En asynchrone, des théorèmes existent certes, mais on n'approche pas du tout la richesse du synchrone. Citons quelques résultats,

- En deux dimensions, sur $\mathbb{Z} \times \mathbb{Z}$, l'automate cellulaire de von Neumann à 29 états, 1949 (232), qui est auto-reproducteur (au même sens que pour les virus informatiques) et Turing-complet (c'est-à-dire capable de calculer toute fonction calculable). Amélioré par E.F. Codd, 1964 (46) (par ailleurs inventeur des bases de données relationnelles) avec 8 états en mimant des fonctionnements de biologie cellulaire.
- Sur $\mathbb{Z}\times\mathbb{Z}$ encore, l'extraordinaire richesse du populaire Jeu de la Vie de John H. Conway (1970) qui est un automate cellulaire à 2 états.
- La synchronisation possible dans n'importe quelle géométrie de graphe de degré borné (Pierre Rosenstiehl, 1966 (188)). De plus, ceci peut se faire avec tolérance aux pannes (Tao Jiang, 1989 (124), cf. aussi (240; 105) pour des améliorations ultérieures).

Le deuxième choix de von Neumann, celui d'une géométrie simple à deux dimensions a été validé a contrario par la malheureuse expérience de la "Connection machine" (1980), basée sur les travaux de Daniel Hillis (117) et à laquelle a participé le prix Nobel Richard Feynman (118). Constituée de 65 536 processeurs, elle a été connectée comme un hypercube de dimension 16. Mais personne n'a su exploiter réellement le parallélisme massif ainsi offert. Est-ce si étonnant ? qui s'y retrouve pour programmer dans un tel hypercube ? Et ce fut un échec commercial qui a gelé pour longtemps l'implémentation du parallélisme massif. Mais aujourd'hui la demande d'un paradigme de programmation massivement parallèle revient avec force, soutenue par les grands noms de l'industrie informatique. Et cette problématique apparaît régulièrement jusque dans des articles du New York Times et du Wall Street Journal... Revenant aux idées de von Neumann avec la géométrie élémentaire, d'autres voies sont aujourd'hui explorées pour forger des outils théoriques permettant de maîtriser le parallélisme massif, cf. J. Mazoyer & J.B. Yunès (160).

1.2.6 Un modèle à part : le Lambda-calcul

Quelques mois plus tôt que Turing, et de façon indépendante, Alonzo Church était arrivé à la conviction d'avoir obtenu un modèle capturant la notion de calcul effectif. Ce avec le λ-calcul qu'il avait développé depuis le début des années 1930. L'article (42) présente son argumentation. Sans pouvoir rentrer de façon approfondie dans le λ-calcul, mentionnons qu'il s'agit d'un langage de termes construit à partir de deux opérations (cf. aussi la Remarque en fin du §1.6.3) :
• *l'application* (t, u) d'un terme t à un autre terme u. Son intuition est celle de l'application d'une fonction à un objet. La difficulté conceptuelle étant qu'en λ-calcul tout est fonction, il

n'y a pas aucun typage [1].
• *l'abstraction* $\lambda x \cdot t$ d'un terme t relativement à une variable x (qui peut figurer ou non dans t). Son intuition est celle de la fonction qui à un objet x associe t (qui peut dépendre de x).
Ce langage de termes est complété par une notion de réduction, la β-réduction, qui remplace l'application d'un terme u à une abstraction $\lambda x \cdot t$ par le terme $t(u/x)$ obtenu en substituant u à x dans t (il y a un petit problème de clash de variable liée sur lequel nous n'insisterons pas). Le λ-calcul est ainsi le premier exemple de système abouti de réécriture. On peut le voir aussi comme une modélisation (assez particulière) de la dualité abstraction/raffinement si importante en génie logiciel.
Comment le λ-calcul peut-il être un modèle de calcul ? Où sont les entiers ? L'idée, très originale, de Church est d'identifier un entier n au terme Church_n qui représente la fonctionnelle qui itère n fois une fonction. Par exemple, le terme Church_3 qui représente l'entier 3 est $\lambda f \cdot \lambda x \cdot (f, f(f, x))$. De la sorte, à chaque terme t on peut associer la fonction sur les entiers qui associe à un entier n l'entier p si le terme (t, Church_n) peut se réduire à Church_p. Church et Kleene ont montré que les termes du λ-calcul représentent exactement les fonctions partielles calculables (cf. §1.3.1).
Mentionnons pour conclure que le λ-calcul est, depuis son origine et encore de nos jours, le modèle "le plus mathématique" de la calculabilité. Il admet une théorie d'une richesse théorique qu'aucun autre modèle n'approche.

1.2.7 La thèse de Church et ses différentes versions

Le fait que tous les modèles considérés aux §1.2.1 à 1.2.6 conduisent à la même notion de calculabilité a été très vite réalisé. Ce qui a conduit à la thèse de Church-Turing, exprimée en fait pour la première fois par Stephen Kleene, alors étudiant d'Alonzo Church :

Ce qui est effectivement calculable *est calculable par une machine de Turing.*

Dans cette formulation, la première notion de *calculable* fait référence à une notion donnée intuitive, alors que la seconde notion de calculable signifie "calculable par une machine de Turing" (90; 54; 171). Les discussions originales de Church, Kleene et Turing sont relatives à des dispositifs algorithmiques de déduction, si l'on veut à la puissance de la déduction formelle, et non pas à la notion de machine ou de dispositif physique de calcul.

Comme l'argumente très justement Jack Copeland dans (54), cette thèse a subi un glissement sémantique historique qui mène souvent à la confondre maintenant avec la thèse suivante, nommée thèse M dans (90), et parfois nommée *version physique de la thèse de Church* :

Ce qui peut être calculé par une machine peut l'être par une machine de Turing.

Dans cette dernière, la notion de machine est intuitive, et est supposée obéir aux lois physiques [2] du monde réel (54). Sans cette hypothèse, la thèse est facile à contredire, cf. les contre-exemples dans (171) ou dans (56). Observons que cette variante de la thèse est intimement reliée au problème de la modélisation de notre monde physique, c'est-à-dire au problème de comprendre si les modèles de notre monde physique sont correctement reliés à ce dernier. En fait, une variante, proche de la dernière thèse est la suivante :

1. Il existe, bien sûr, des versions typées du λ-calcul, mais celui considéré par Church n'est pas typé.
2. Sinon à ses contraintes sur les ressources.

Tout processus qui admet une description mathématique peut être simulé par une machine de Turing (54).

Encore une fois (et pour globalement les même contre-exemples en fait que pour la thèse physique), si le processus n'est pas contraint de satisfaire les lois physiques de notre monde réel alors cette thèse est facile à contredire (54).

Ces trois thèses sont indépendantes : la première concerne la puissance des systèmes formels, comme les systèmes de déduction ou de preuve (54) ; la seconde concerne la physique du monde qui nous entoure (215; 54; 239) ; la troisième concerne les modèles que nous avons du monde qui nous entoure (215; 54; 239).

Chacune de ces thèses, faisant référence à une notion intuitive, ne saurait être complètement prouvée [3]. Il peut toutefois être intéressant de discuter ces thèses.

D'une part, il est possible de chercher à définir un certain nombre d'axiomes minimaux que doivent satisfaire un système formel, ou une machine physique pour rendre ces thèses prouvables. Cela permet de réduire ces thèses à des hypothèses minimales sur les systèmes considérés, et peut aider à se convaincre de leur validité (90; 67; 27).

D'autre part, si l'on prend chacune de ces thèses de façon contraposée, chacune signifie que tout système qui calcule quelque chose non calculable par une machine de Turing doit utiliser un ingrédient, que nous appellerons *ressource*, qui soit n'est pas calculable algorithmiquement, pour la première, soit n'est pas calculable par une machine physique, pour la seconde, soit n'est pas calculable par un modèle de machine physique, pour la troisième. Dans tous les cas, on peut qualifier (si besoin par définition) une telle ressource de "non-raisonnable". Il peut alors sembler important de discuter ce qui fait qu'une ressource peut être non raisonnable, indépendamment de la véracité de chacune des thèses, pour mieux comprendre le monde et les modèles du monde qui nous entourent.

Enfin, on peut observer que ces thèses expriment des faits très profonds sur la possibilité de décrire par les mathématiques ou par la logique le monde qui nous entoure (72), et plus globalement sur les liens entre calculabilité, mathématiques et physique :
- la nature calcule-t-elle ?
- quels sont les liens entre non-déterminisme, chaos, non-prédictibilité, et aléatoire (151) ?

1.2.8 Axiomatisation de la thèse de Church par Gandy

Robin Gandy propose dans (90) de réduire la thèse physique de Church à quatre principes (assez techniques). L'intérêt de cette axiomatisation est que l'on peut alors prouver mathématiquement que ce qui est calculable par un dispositif physique satisfaisant à ces quatre principes est calculable par machine de Turing. Autrement dit, Gandy propose de remplacer la thèse physique de Church par la thèse qu'un système mécanique physique déterministe discret satisfait à ces quatre principes. Gandy suppose que l'espace physique est l'espace géométrique ordinaire à trois dimensions. Suivant (72), on peut dire que l'argumentation de Gandy traduit dans ces quatre principes deux grandes hypothèses relatives à la nature physique : la finitude de la vitesse de la transmission de l'information et la finitude de la densité de l'information. La discussion de Gandy se limite aux systèmes discrets digitaux, et écarte explicitement les

3. En fait, la notion de calcul, ou de machine, est souvent définie dans nombre d'ouvrages de calculabilité ou de complexité, en supposant ces thèses (ou l'une de ces thèse) comme vraie.

systèmes analogiques dès son début. Développée pour les systèmes déterministes, elle peut s'étendre aux systèmes non-déterministes assez facilement.

Les principes de Gandy sont satisfaits par tous les modèles de calcul discrets usuels, y compris ceux qui sont massivement parallèles comme le jeu de la vie (cf. §1.2.5). En revanche, les deux hypothèses physiques mentionnées plus haut, ne sont pas nécessairement vraies dans toute théorie physique. Il peut alors être intéressant de comprendre ce qui se passe par exemple en mécanique quantique, ou en physique relativiste. Pour la physique quantique, nous renvoyons à (169) pour une discussion des sources de non-calculabilité, et les conséquences à en tirer sur la thèse de Church ou sur nos modèles physiques. L'article (6) discute par ailleurs de principes à la Gandy qui permettraient de capturer la théorie des calculs quantiques.

Mentionnons que l'axiomatisation de Gandy a été simplifiée et étendue ultérieurement par différents travaux de Wilfried Sieg : voir par exemple (208; 206; 207; 209). Une toute autre axiomatisation de la notion d'algorithme, due à Gurevich, sera vue en §1.4.

1.3 Théorie de la calculabilité

Nous avons vu plus haut la diversité des modèles de calcul et le fait remarquable que tous conduisent à la même classe de fonctions calculables. Fait qui conduit à la thèse de Church assurant que chacun d'eux est une formalisation de la notion intuitive de calculabilité.

1.3.1 Calculs qui ne s'arrêtent pas

Un autre fait important est que, dans chacun de ces modèles, il y a des calculs qui ne s'arrêtent pas. Comme chacun le sait, il n'y a pas non plus de langage de programmation des seules fonctions calculables. Il y a toujours des programmes qui ne terminent pas dans certains cas, expérience banale de tout programmeur...

D'où la notion de fonction partielle calculable (ou fonction récursive partielle) : *fonction non partout définie dont la valeur là où elle est définie est donné par le résultat d'un algorithme.* La dénomination "fonction partielle calculable" n'est pas très heureuse car le qualificatif partiel a un double sens : d'une part, il exprime que la fonction est partielle, c'est-à-dire non partout définie, d'autre part, la calculabilité de cette fonction est partielle car elle ne permet pas de décider si la fonction est ou non définie en un point mais seulement de connaître sa valeur lorsqu'elle est définie. Daniel Lacombe, dans les années 1960, avait essayé (sans succès) d'introduire la dénomination bien meilleure de "semi-fonction semi-calculable" ou de "fonction partielle partiellement calculable" (cf. (140), §1.9).

La contrepartie ensembliste de cette notion de fonction partielle calculable est celle d'ensemble calculablement énumérable (ou récursivement énumérable), c'est-à-dire *d'ensemble pour lequel il existe une énumération des éléments qui est calculable*. Ainsi, si un élément est dans l'ensemble, on finit par le savoir. En revanche, si un élément n'y est pas, on n'a a priori aucun moyen de le savoir.

Ce phénomène de calculabilité partielle peut apparaître comme une scorie de la calculabilité, en fait, il s'agit d'un trait fondamental qui permet une vraie théorie de la calculabilité. Il est important de le souligner : *Malgré le grand nombre de livres intitulés "théorie de la calculabilité", il n'y a pas de théorie significative de la calculabilité. En revanche, il y a une remarquable théorie de la calculabilité partielle.* Citons quelques résultats spectaculaires que nous énonçons

pour les fonctions sur les entiers, mais on pourrait le faire pour les fonctions sur toute autre famille d'objets finis : mots, arbres ou graphes finis...

1.3.2 Trois théorèmes "positifs" en calculabilité partielle

Théorème d'énumération. Adoptons la convention d'écriture $f(e,x) = f_e(x)$. Il existe une fonction partielle calculable $\varphi^{(1)} : \mathbb{N}^2 \to \mathbb{N}$ telle que la famille des fonctions $x \mapsto \varphi_e^{(1)}(x)$, $e \in \mathbb{N}$, soit exactement la famille des fonctions partielles calculables $\varphi : \mathbb{N} \to \mathbb{N}$. Même chose pour les fonctions avec k variables et une fonction $\varphi^{(k)} : \mathbb{N}^{1+k} \to \mathbb{N}$. De manière plus intuitive, la fonction $\varphi^{(1)}$ correspond à la définition d'un *interpréteur*. Ainsi, $\varphi_e^{(1)}(x)$ est l'évaluation du programme e sur l'entrée x. Le livre de Jones (127) fait le pont entre la calculabilité et la théorie de la programmation.

Théorème du paramètre (ou s-m-n) (Kleene, 1943). Il existe une fonction calculable totale $s : \mathbb{N}^3 \to \mathbb{N}$ telle que, pour toute fonction partielle calculable $f : \mathbb{N}^2 \to \mathbb{N}$ de programme e, et tout x et y, on a

$$\varphi^1_{s(e,y)}(x) = \varphi^2_e(x,y)$$

La fonction s calcule un nouveau programme à partir d'un programme existant e et d'une entrée y. Ce résultat a des conséquences d'une surprenante richesse. Tout d'abord, il dit qu'un programme et une entrée sont interchangeables : la grande idée de John von Neumann pour les ordinateurs ! D'autre part, s spécialise un paramètre du programme e par y, et en cela le théorème du paramètre de Kleene est la pierre de base de l'évaluation partielle. Au passage, mentionnons la projection de Futamura-Ershov-Turchin qui construit un compilateur en spécialisant un interpréteur d'un langage de programmation avec le programme à compiler. On se reportera encore aux travaux de Jones (127).

Théorème du point fixe de Kleene (1938). Quelle que soit la fonction partielle calculable $f : \mathbb{N} \to \mathbb{N}$, il existe e telle que, pour tout x, $\varphi_e^{(1)}(x) = f(e,x)$. Ainsi, pour toute transformation de programme, il y a un programme qui fait la même chose que son transformé. Evidemment, en général ce programme (ainsi que son transformé) ne fait rien, c'est-à-dire ne termine pas. Mais ce n'est pas toujours le cas : ce théorème est, de fait, un des outils les plus puissants de la calculabilité. Ce théorème a de nombreuses conséquences et il permet de montrer en général des résultats négatifs d'indécidabilité. Les liens avec la logique ont été explorés, avec le brio qu'on lui connaît, par Smullyan (217; 218). Ceci étant, un exemple amusant d'application positive est l'obtention d'un programme sans entrée dont la sortie est le programme lui-même ! De tels programmes sont appelés des "quines". Ce théorème du point fixe est aussi la source d'une théorie de la virologie, qui a débuté avec la thèse de Cohen (47) et l'article de son directeur Adleman (3). L'implication du théorème du point fixe dans la virologie a été éclaircie dans (28). Enfin, ce résultat est aussi employé en apprentissage (123).

Après ces théorèmes positifs, voyons deux théorèmes catastrophes.

1.3.3 Deux théorèmes "négatifs" en calculabilité partielle

Indécidabilité de l'arrêt. Le domaine de $\varphi^{(1)}$ n'est pas calculable. On ne peut pas décider par algorithme si le calcul sur une entrée donné s'arrête ou non. Théorème qui peut aussi être

vu comme théorème anti-chômage : on aura toujours besoin d'un chercheur pour essayer de décider ce genre de question, pas possible de le remplacer par un programme.

Théorème de Rice (1954), version programmes. Quelle que soit la famille $\mathcal{P}$ de fonctions partielles calculables, si $\mathcal{P}$ n'est ni vide ni pleine (il y a une fonction partielle calculable non dans $\mathcal{P}$) alors $\{e \mid \varphi_e^{(1)} \in \mathcal{P}\}$ (qui est l'ensemble des programmes qui calculent une fonction dans $\mathcal{P}$) n'est pas un ensemble calculable. Ainsi, *on ne peut décider aucune question non triviale sur la sémantique des programmes*.

1.4 Formalisation de la notion d'algorithme

1.4.1 Dénotationnel versus opérationnel

Qu'est-ce qu'un algorithme. Si la formalisation de la notion intuitive de calculabilité est faite vers 1936, la question de savoir ce qu'est un algorithme reste cependant ouverte. En effet, les très nombreux résultats étayant la thèse de Church montrent qu'il y a bien des formalisations de classes d'algorithmes (ceux associés aux machines de Turing, aux machines RAM, aux programmes C, etc.) qui suffisent à calculer toutes les fonctions (partiellement) calculables. Mais ces classes sont clairement distinctes : un programme à la Kleene avec boucles `FOR` et `WHILE` peut mimer et être simulé par un programme de machine RAM ou un automate cellulaire mais il n'a rien de commun avec (sauf l'association entrée/sortie). Ce sont donc des résultats de *complétude dénotationnelle*. Mais y a-t-il une notion formelle d'algorithme englobant toutes ces classes ? En d'autres termes peut-on obtenir une classe formelle d'algorithmes qui soit *opérationnellement complète* ? Soulignons que la thèse de Church ne dit rien à ce sujet : elle est relative au dénotationnel pas à l'opérationnel.

Les tentatives de Kolmogorov & Uspensky et de Schönhage. Le premier à s'attaquer à ce problème est Kolmogorov vers 1953 (136; 137). C'est dans ce but qu'il semble avoir introduit les machines de Kolmogorov-Uspenski (cf. §1.2.2). Mais il ne dispose pas d'outil lui permettant d'argumenter dans ce sens. En fait, on sait depuis les travaux de Yuri Gurevich, (71), que ni ce modèle ni celui des machines de Schönhage (cf. §1.2.2) ne sont opérationnellement complets. Cependant, ils le deviennent si on leur rajoute une fonction de codage des couples (une injection de $\mathbb{N}^2$ dans $\mathbb{N}$) !

1.4.2 Complétude opérationnelle

Les ASM de Yuri Gurevich. Vers 1984, Yuri Gurevich (112; 113). introduit une notion de machine complètement nouvelle, basée sur la logique : les machines abstraites à états (en anglais "Abstract State Machines", originellement appelées "Evolving Algebras"). Une ASM est une structure munie de fonctions (les relations sont confondues avec des fonctions à valeurs Booléenes) :

- Certaines fonctions sont fixes. Elles constituent la partie statique de l'ASM et correspondent aux primitives de programmation (ou encore à une bibliothèque).
- D'autres peuvent évoluer au cours du calcul selon un programme écrit dans un langage qui ne comprend que des affectations, la conditionnelle et une notion de bloc parallèle.

C'est la partie dynamique de l'ASM qui traduit l'environnement (par nature dynamique) du programme.

Ce modèle est extrèmement flexible et, en variant partie statique et vocabulaire dynamique, on arrive à simuler "pas à pas" tous les modèles de calcul séquentiel connus (cf. l'abondante littérature citée sur la page web de Gurevich). On peut même améliorer cette simulation "pas à pas" en "identité littérale" (106).

Complétude opérationnelle des ASM. Ces résultats de simulation pas à pas justifient la *thèse de Gurevich (ou thèse du calcul séquentiel)*, énoncée dès la première publication, 1985 (110), de Gurevich sur ce modèle :

Toute façon "séquentielle" de calculer est simulable pas à pas *par une ASM.*

C'est le premier résultat de complétude opérationnelle !
Mentionnons que Gurevich a aussi étendu ses travaux aux algorithmes parallèles et aux algorithmes concurrents, notions plus délicates encore (20; 21).

Complétude opérationnelle du Lambda-calcul. Un travail récent, 2009 (84), montre que le comportement d'une ASM est simulable en lambda calcul, une étape de calcul de l'ASM étant simulée par un groupe de k réductions successives en λ-calcul, où k est un entier fixe ne dépendant que de l'ASM considérée et pas des entrées. Le λ-calcul est donc aussi opérationnellement complet.

L'approche de Moschovakis. Une autre approche d'un modèle opérationnellement complet a été entreprise par Yannis Moschovakis (166) en utilisant la notion de définition par plus petit point fixe. Elle rend compte de la notion d'algorithme à un niveau plus abstrait que celui des ASM de Gurevich.

1.4.3 L'axiomatisation de la notion d'algorithme par Gurevich

Yuri Gurevich propose d'axiomatiser la notion d'algorithme par les trois axiomes suivants :

i. Un algorithme détermine une suite "d'états de calcul" pour chaque entrée valide.

ii. Les états d'une suite d'états de calcul sont des structures d'un même langage logique qui ont toutes le même domaine.

iii. Il existe une famille finie de termes du langage telle que le successeur d'un état quelconque ne dépende que des interprétations de ces termes.

Cette axiomatisation ainsi que sa justification est évidemment intimement liée à la théorie des ASM. Gurevich, 2000 (113), prouve formellement sa thèse à partir de ces trois axiomes.

Thèse de Church et thèse de Gurevich. Rajoutant un quatrième axiome,

iv. Seules des opérations considérées comme indéniablement calculables sont disponibles dans les états initiaux.

Nachum Dershowitz et Yuri Gurevich, 2008 (67) (qui font suite à (27)), prouvent que tout processus calculatoire qui satisfait ces quatre axiomes vérifie la thèse de Church. La beauté de cette axiomatisation (67) est d'être générique, formelle, et basée sur des formalismes communément admis. D'autre part, elle permet de réduire prouvablement tout doute sur sa validité en doute sur l'un de ces quatre principes élémentaires.
Mentionnons qu'il est possible d'étendre ces résultats à la notion d'algorithme parallèle (20), et de la relier à la notion d'algorithme quantique (103).

1.5 Calculabilité sur les réels

Il existe différents modèles pour capturer les calculs sur les réels, modèles issus de motivations très différentes.

1.5.1 Analyse récursive

Née des travaux de Turing (224), Grzegorczyk (108), et Lacombe (139), l'analyse récursive considère qu'un réel α est calculable s'il est possible d'en produire algorithmiquement une représentation, c'est-à-dire une suite d'approximations rationnelles qui converge rapidement vers ce réel, c'est-à-dire encore une suite $(q_n)_{n\in\mathbb{N}}$ de rationnels telle que $|q_n - \alpha| \leq 2^{-n}$ pour tout n (on peut remplacer $(2^{-n})_{n\in\mathbb{N}}$ par toute autre suite récursive $(\varepsilon_n)_{n\in\mathbb{N}}$ de rationnels qui tend vers 0).

Une fonction est calculable s'il est possible de produire algorithmiquement une représentation de son image, à partir de n'importe quelle représentation de son argument.

Attention, la simplicité de cette définition cache une difficulté liée à la représentation discrète du continu. Il est facile de montrer que, quelle que soit la base $p \geq 2$, un réel α est récursif si et seulement si sa "partie fractionnaire" $\alpha - \lfloor\alpha\rfloor$ (qui est dans l'intervalle $[0, 1[$) peut se représenter par un mot infini récursif en base p. Mais ceci ne passe pas aux fonctions récursives sur les réels. Par exemple, on ne peut pas calculer par algorithme la suite des décimales en base 10 de $3x$ à partir de celle de x. On peut seulement calculer une suite d'approximations de $3x$. Ainsi, la représentation par suite de Cauchy récursive et rapide est essentielle pour la notion de fonction calculable sur les réels.

Ce phénomène est un avatar d'un fait (négatif) très important de l'analyse récursive.

Théorème de Rice (1954), version réels. *On ne peut pas décider si un réel est nul.*

1.5.2 Modèle de Blum, Shub et Smale

Le modèle de Blum, Shub et Smale (23; 22) a été introduit comme un modèle pour discuter de la complexité algébrique de problèmes sur les réels. Dans celui-ci, on se fixe des opérations arithmétiques réalisables en temps unitaire, et la complexité d'un problème est mesurée en termes du nombre d'opérations. Notons que, parmi les opérations admises, il y a le test à zéro d'un réel. Ce modèle est donc incompatible avec l'analyse récursive...

1.5.3 "General Purpose Analog Computer" de Shannon

Le General Purpose Analog Computer a été introduit par Claude Shannon, 1941 (202), comme un modèle des machines analogiques de son époque, en particulier de l'Analyseur Différentiel de Vannevar Bush (38) et des circuits analogiques électroniques. Dans la version raffinée actuelle, cf. (101), les unités de base d'un GPAC donnent les fonctions constantes, l'addition, la multiplication et les opérateurs d'intégration $(u, v) \mapsto w$ où $w(t) = \int_{t_0}^{t} u(x)v'(x)dx$ (avec t_0 quelconque). Ces unités peuvent être connectées librement dans un circuit avec retours arrières pour l'intégration (feedback connexions). Une fonction est GPAC-engendrée si on peut la lire à la sortie d'un tel circuit, cf. Figure 1. Les fonctions GPAC-engendrées sont exactement les fonctions différentiellement algébriques, c'est-à-dire les solutions d'équations

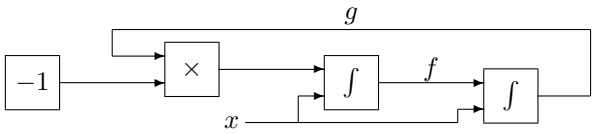

FIGURE 1 – *Un GPAC engendrant les fonctions* $f(x) = \cos(x)$ *et* $g(x) = \sin(x)$

différentielles $p(x, y, y', y'', \ldots, y^{(k)}) = 0$ où p est un polynôme (Shannon (202), voir Graça (101)). Ce sont aussi les composantes des solutions $\vec{y} = (y_1, \ldots, y_k)$ des systèmes différentiels $\vec{y'} = p(t, \vec{y})$, $\vec{y}(0) = \vec{a}$, où p est un k-uplet de polynômes et $\vec{a}$ un k-uplet de réels (102).
Ces fonctions GPAC-engendrables n'incluent pas toutes celles de l'analyse récursive : manquent, par exemple, la fonction $\zeta(x) = \sum_{n \geq 1} \frac{1}{n^x}$ de Riemann et la fonction $\Gamma(x) = \int_0^{+\infty} t^{x-1} e^{-t} dt$, laquelle étend aux réels (même aux complexes) la fonction factorielle $n \mapsto n! = 1 \times 2 \times \ldots \times n$.
On peut cependant retrouver exactement les fonctions de l'analyse récursive (Bournez, Campagnolo, Graça & Hainry, 2007 (31)) comme celles qui sont GPAC-calculables au sens suivant (Graça, 2004 (100)) :
• un GPAC est récursif si les réels des unités constantes et des bornes inférieures des unités d'intégration sont tous récursifs,
• $f : \mathbb{R} \to \mathbb{R}$ est GPAC-calculable s'il existe deux fonctions g, ε engendrées par un GPAC récursif et telles que, pour tout x, on ait $\forall t \ |f(x) - g(t, x)| \leq \varepsilon(t, x)$ et $\lim_{t \to +\infty} \varepsilon(t, x) = 0$.

1.5.4 Modèles de calcul à temps continu

Nous renvoyons à (29) pour un panorama des différents modèles de calcul à temps continu et de leurs propriétés du point de vue de la calculabilité ou de la complexité. En particulier, les modèles issus des réseaux de neurones formels (211), modèles de calcul dont la conception est très schématiquement inspirée du fonctionnement de vrais neurones (humains ou non)).
Mentionnons que, dans plusieurs de ces modèles, il est possible d'exploiter le fait que l'on peut coder une suite *non calculable* dans un réel, ou d'utiliser le fait que l'on suppose une arithmétique exacte, pour réaliser des calculs super-Turing : voir par exemple (210) pour les réseaux de neurones.

1.5.5 Le problème de l'unification de ces modèles

En particulier, dans plusieurs de ces modèles, il est possible d'exploiter le fait que l'on peut coder une suite (non-calculable) dans un réel, ou utiliser le fait que l'on suppose une arithmétique exacte pour réaliser des calculs super-Turing : voir par exemple (210) pour les réseaux de neurones.

Il est clairement impossible d'unifier toutes les approches : par exemple toute fonction calculable en analyse récursive est nécessairement continue, alors que des fonctions discontinues sont calculables dans le modèle initial de Blum Shub et Smale (23). Cependant, si l'on met de coté l'approche algébrique du modèle de Blum Shub et Smale, certains résultats récents établissent l'équivalence entre différents modèles (voir par exemple (30; 31; 102)). Il n'est pas cependant clair à ce jour qu'il puisse exister un concept unificateur pour les modèles continus

analogue à la thèse de Church pour les modèles discrets/digitaux : nous renvoyons au survol (29) pour des discussions sur le sujet.

1.5.6 Calculabilité au-delà de la thèse de Church ?

Aussi, il peut être intéressant de suivre (171), et de faire un rapide panorama (non-exhaustif) de différents moyens d'obtenir des systèmes ou des machines plus puissants que les machines de Turing.

Calculabilité avec oracle. La première façon, proposée par Turing lui-même dans (225) consiste à considérer des machines avec oracles, qui correspondent à des boites noires capables de répondre à certaines questions. Cela permet classiquement de "relativiser" nombre de résultats en calculabilité et complexité, mais aussi d'obtenir des modèles plus puissants que les machines de Turing (dès que l'oracle n'est pas calculable).
Il y a plusieurs types de machines ou de systèmes dans l'esprit des machines à oracles. Cela inclut les machines de Turing couplées de (53), qui sont des machines de Turing connectées à un canal d'entrée, ou les réseaux de machines qui exploitent le fait qu'un temps de synchronisation peut être irrationnel et non-calculable (56), ou les machines couplées à des dispositifs physiques comme les *scatter-machines* de (11), ou utilisant des tirages aléatoires biaisés comme dans (172).

Machines accélérantes. Une seconde approche consiste à considérer des machines accélérantes, c'est-à-dire à considérer des machines qui effectuent leur première opération en temps unitaire, et chaque opération ultérieure en un temps égal à la moitié de l'opération précédente. Cette idée, ancienne, déjà présente dans (19; 191; 236), est aussi présente dans les machines à temps ordinal : voir par exemple (115). Il a été argumenté que différents systèmes physiques pouvaient en réaliser des implémentations : voir par exemple (40) à propos de la mécanique quantique, ou (119) à propos de calculs par trous noirs.

Autres machines. Il est aussi possible de considérer différentes variantes des machines de Turing : des machines exploitant un non-déterminisme non borné (220), travaillant sur des entrées infinies (voir par exemple (234) pour une présentation des machines de Turing de type 2), des machines avec une infinité d'états (170), des machines bruitées (7), etc…

1.5.7 Calculabilité quantique

Une machine de Turing quantique. Bien que Feynman ait pressenti l'intérêt d'un ordinateur quantique (85; 86), le premier modèle de calcul quantique est la machine de Turing quantique (MTQ) introduite par Deutsch (68) en 1985. Cette machine de Turing est une extension d'une machine probabiliste, ou chaque transition s'effectue avec une amplitude (un nombre complexe) donnée. Les MTQ permettent de rendre compte de phénomènes quantiques comme la superposition, les interférences et l'intrication. Bernstein et Vazirani (15) ont prouvé l'existence d'une MTQ universelle, c'est-à-dire capable de simuler efficacement de façon approchée toute MTQ. Yao (238) a montré que les circuits quantiques, introduit par Deutsch (69), sont polynomialement équivalents aux machines de Turing quantiques.

Algorithmes et complexité quantiques. Deutsch (68), puis Deutsch et Jozsa (70) ont démontré l'intérêt du calcul quantique en introduisant un algorithme quantique qui permet de décider

si une fonction $f : \{0,1\}^n \rightarrow \{0,1\}$ est constante ou équilibrée ($|f^{-1}(0)| = |f^{-1}(1)|$) en un seul appel à l'oracle f, alors que tout algorithme classique déterministe pour ce problème nécessite $2^{n-1}+1$ appels. Cet algorithme ne présente pas d'application pratique, mais il est la première preuve de la supériorité du calcul quantique sur le calcul classique. Plus utile, l'algorithme de Grover (107) permet de trouver un élément dans une base de données non structurée de taille n en $O(\sqrt{n})$ requêtes, alors que $\Omega(n)$ requêtes sont nécessaires à un algorithme classique.

La principale classe de complexité quantique est BQP (15). Elle est définie par analogie à la classe probabiliste BPP. BQP (resp. BPP) est la classe des problèmes de décision qui peuvent être résolus en temps polynomial par une machine de Turing quantique (resp. probabiliste) avec une probabilité d'erreur inférieure à 1/3. Comme $PTIME \subseteq BPP \subseteq BQP \subseteq PSPACE$ (15), prouver une séparation entre BPP et BQP est une question considérée comme difficile car elle permettrait de résoudre le problème ouvert $PTIME =^{?} PSPACE$.

Pourtant plusieurs algorithmes quantiques, comme l'algorithme de Simon (212) et celui de Shor (205), semblent témoigner d'une telle séparation : l'algorithme de Shor permet la factorisation et le calcul du logarithme discret en temps polynomial, alors qu'aucun algorithme classique polynomial n'est connu pour résoudre ces problèmes.

En revanche, aucun algorithme quantique polynomial pour résoudre un problème NP-complet n'est connu, et les classes NP et BQP sont à l'heure actuelle incomparables.

1.6 Déduction et calcul : la nature algorithmique des preuves constructives

1.6.1 Les preuves constructives comme langage de programmation

Un des résultats les plus étonnants de l'informatique théorique et de la programmation est le lien profond qui existe entre déduction et calcul.

La propriété du témoin. L'illustration la plus simple de la nature algorithmique des preuves est donnée par la *propriété du témoin* des preuves "constructives" [4] :

> *D'une preuve constructive d'une formule de la forme $\exists y\ A$, il est possible d'extraire un terme u et une preuve constructive de la formule $A(u/y)$.*

De la même façon, d'une preuve constructive d'une formule exprimant l'existence d'une fonction, c'est-à-dire d'une formule de la forme $\forall x \exists y\ A$, il est possible d'extraire un programme calculant une telle fonction : à partir d'un terme t (*l'entrée du programme*) et de la preuve constructive de la formule $\forall x \exists y\ A$, on peut (par instanciation) trouver une preuve constructive de la formule $\exists y\ A(t/x)$ et donc un terme u (*la sortie du programme*) ainsi qu'une preuve constructive de la formule $A(t/x, u/y)$. Ainsi, le langage des preuves constructives peut-il être vu comme un langage de programmation dont le mécanisme d'exécution des programmes coïncide avec celui de l'extraction du témoin dans les preuves, mécanisme appelé aussi *réduction des preuves.*

Un langage de programmation avec spécification prouvée. Un avantage de ce langage des

4. On dit aussi preuve "intuitionniste". Nous préférons le qualificatif "constructive" car il met l'accent sur le caractère opérationnel, algorithmique donc, alors que le qualificatif "intuitionniste" relève du vocabulaire philosophique.

preuves constructives sur les autres langages de programmation est, qu'en plus de la sortie u, le mécanisme d'extraction fournit une preuve de la formule $A(t/x, u/y)$, c'est-à-dire une preuve de ce que t and u satisfont bien la spécification A.
On imagine combien ce peut aussi être un outil puissant en intelligence articielle pour la recherche de preuve de spécification.

C'est aussi **un langage de programmation dont tout programme termine.** Certes, cette propriété interdit au langage d'être Turing complet : d'une part, il ne permet pas de programmer toutes les fonctions calculables partout définies (il en manquera), d'autre part, pour une fonction qu'il permet de calculer, il ne pourra pas en implémenter tous les algorithmes de calcul possibles. Néanmoins, ce langage des preuves constructives (tout particulièrement dans ses versions étendues aux théories, cf. §1.6.5) est d'une grande richesse expressive, contrairement aux autres langages de programmation dans lesquels tout programme termine (comme le langage de la récursivité primitive avec la seule boucle `FOR`, cf. §1.2.1). La raison en est qu'il permet d'exprimer toutes les fonctions dont l'existence peut être prouvée dans la théorie considérée. En effet, si on peut prouver l'existence d'une fonction dans cette théorie, alors cette preuve d'existence est elle-même un programme de calcul de cette fonction (dans ce langage des preuves)...

Pour aller plus loin dans cette utilisation des langages de preuves comme langages de programmation, voir l'article de Christine Paulin-Mohring et Benjamin Werner, 1993 (179).

Remarque. Nous avons insisté sur l'aspect langage de programmation de ce langage des preuves constructives. Mais, sans chercher à le structurer en langage de programmation, on peut aussi le voir comme un outil puissant de spécification des plus utiles en IA.

1.6.2 Les preuves constructives et leurs notations

La déduction naturelle constructive (ou intuitionniste)

Usuellement, définir une notion de prouvabilité, c'est définir la famille des formules qui seront dites prouvables. Cependant, pour obtenir le langage des preuves mentionné plus haut, on ne s'appuie pas sur cette manière usuelle de faire, laquelle ne serait pas très commode, mais sur un système introduit par Gerhard Gentzen, 1934 (92) : *la déduction naturelle constructive (ou intuitionniste)*. Comme son nom l'indique, ce système suit de très près les façons naturelles de faire des preuves, c'est-à-dire les façons de faire des mathématiciens. En particulier, pour prouver une formule de la forme $A \Rightarrow B$, ce que fait un mathématicien est de supposer A et de prouver alors B en se servant de cette hypothèse A. Ceci conduit à définir non pas directement la famille des formules prouvables mais une famille de couples constitués d'une liste finie de formules hypothèses (appelée le contexte) et d'une formule conclusion. Un tel couple est appelé un *séquent* et est noté $A_1, ..., A_n \vdash B$, la liste $(A_1, ..., A_n)$ étant le contexte, et la formule B la conclusion. Les axiomes et les règles de déduction de la déduction naturelle sont donnés par la Table 1 [5]. On notera que la règle d'introduction du $\Rightarrow$ correspond bien à ce qu'on a dit plus haut de la manière de faire du mathématicien. Un examen attentif de ces règles montre que chacune correspond à une façon de raisonner tout à fait naturelle et usuelle. Notons enfin

5. Dans cette table, les axiomes ont été représentés comme des règles sans prémisse. D'autre part, les lettres A, B, C représentent des formules quelconques et la lettre t représente un terme quelconque. Il s'en suit que chaque axiome est en fait une liste infinie d'axiomes et chaque règle une liste infinie de règles.

$\dfrac{}{A_1, ..., A_n \vdash A_i}$ axiome	
$\dfrac{\Gamma \vdash \bot}{\Gamma \vdash A}$ $\bot$-élim	$\dfrac{}{\Gamma \vdash \top}$ $\top$-intro
$\dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}$ $\dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}$ $\wedge$-élim	$\dfrac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$ $\wedge$-intro
$\dfrac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}$ $\vee$-élim	$\dfrac{\Gamma \vdash A}{\Gamma \vdash A \vee B}$ $\dfrac{\Gamma \vdash B}{\Gamma \vdash A \vee B}$ $\vee$-intro
$\dfrac{\Gamma \vdash A \quad \Gamma \vdash A \Rightarrow B}{\Gamma \vdash B}$ $\Rightarrow$-élim	$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}$ $\Rightarrow$-intro
$\dfrac{\Gamma \vdash \forall x\ A}{\Gamma \vdash A(t/x)}$ $\forall$-élim	$\dfrac{\Gamma \vdash A}{\Gamma \vdash \forall x\ A}$ $\forall$-intro (si x n'est pas libre dans Γ)
$\dfrac{\Gamma \vdash \exists x\ A \quad \Gamma, A \vdash B}{\Gamma \vdash B}$ $\exists$-élim (si x n'est pas libre dans Γ, B)	$\dfrac{\Gamma \vdash A(t/x)}{\Gamma \vdash \exists x\ A}$ $\exists$-intro

TABLE 1 – *Axiomes et règles en déduction naturelle constructive (ou intuitionniste)*

que les axiomes sont triviaux puisqu'il s'agit des séquents dont la conclusion figure dans le contexte... Le lecteur peut s'étonner de ne pas y trouver de règle pour la négation. La raison en est que l'on considère que la négation $\neg A$ est la formule $A \Rightarrow \bot$, c'est-à-dire la formule qui exprime que A implique le faux.

A partir de ces axiomes et règles, on définit une preuve comme un arbre dont les sommets sont étiquetés par des séquents de sorte que
(i) les feuilles sont étiquetées par des axiomes,
(ii) chaque noeud interne ayant N fils (où $N = 1, 2, 3$) est étiqueté par une règle ayant N prémisses.
Un exemple de preuve est le suivant

$$\dfrac{\dfrac{\dfrac{\dfrac{}{p \Rightarrow (q \Rightarrow r), q, p \vdash p \Rightarrow (q \Rightarrow r)}\text{ax} \quad \dfrac{}{p \Rightarrow (q \Rightarrow r), q, p \vdash p}\text{ax}}{p \Rightarrow (q \Rightarrow r), q, p \vdash q \Rightarrow r}\Rightarrow\text{-él} \quad \dfrac{}{p \Rightarrow (q \Rightarrow r), q, p \vdash q}\text{ax}}{p \Rightarrow (q \Rightarrow r), q, p \vdash r}\Rightarrow\text{-él}}{p \Rightarrow (q \Rightarrow r), q \vdash p \Rightarrow r}\Rightarrow\text{-intro}}{p \Rightarrow (q \Rightarrow r) \vdash q \Rightarrow (p \Rightarrow r)}\Rightarrow\text{-intro}$$

Les séquents prouvables sont les séquents qui étiquetent les preuves. Remarquons aussi que les arbres ne vérifiant que la condition (ii) correspondent à des règles dérivées dont les prémisses sont toutes les feuilles qui ne sont pas des axiomes. Rajouter de telles règles dérivées n'augmente pas la famille des séquents prouvables.

Enfin, le cadre usuel de prouvabilité se retrouve comme suit :
- une preuve d'une formule A est une preuve du séquent de contexte vide $\varnothing \vdash A$,
- une preuve d'une formule A dans une théorie axiomatique $\mathcal{T}$ (c'est-à-dire un ensemble $\mathcal{T}$ de formules) est une preuve d'un séquent $\Gamma \vdash A$ dont le contexte Γ est inclus dans $\mathcal{T}$.

Remarque. Comme on est en logique constructive, certaines propriétés "classiques" ne sont plus prouvables. Nous citons ci-dessous trois exemples de tautologies de la logique "classique" (le premier est la formule de Pierce, les deux autres traduisent les formules $p \vee \neg p$ et $\neg\neg p \Rightarrow p$) dont aucune n'admet de preuve constructive :

$$((p \Rightarrow q) \Rightarrow p) \Rightarrow p \qquad , \qquad p \vee (p \Rightarrow \bot) \qquad , \qquad ((p \Rightarrow \bot) \Rightarrow \bot) \Rightarrow p$$

Contextes

L'étiquetage des noeuds des arbres de preuves par des séquents est plutôt lourd car les contextes sont des listes de formules qui se répètent de noeud en noeud. Une notation plus légère serait bienvenue. L'idée la plus simple est d'oublier les séquents et de ne garder en étiquettes que les noms des règles utilisées. Mais cela ne suffit pas car il faut bien indiquer les formules en jeu dans la règle. Comme les contextes varient assez peu dans une preuve, on procède comme suit.

• On introduit un nom α_i pour chaque hypothèse A_i du contexte du séquent de la racine de l'arbre de preuve. On obtient ainsi le *contexte nommé* $(\alpha_1{:}A_1, ..., \alpha_n{:}A_n)$. Pour l'utilisation d'un axiome, il suffit alors d'indiquer le nom de l'hypothèse en jeu. Ainsi, avec le contexte

nommé $(\alpha{:}P \Rightarrow Q, \beta{:}P)$ pour la racine, la preuve

$$\dfrac{\dfrac{}{\alpha{:}P \Rightarrow Q, \beta{:}P \vdash P \Rightarrow Q}\ \text{axiome} \qquad \dfrac{}{\alpha{:}P \Rightarrow Q, \beta{:}P \vdash P}\ \text{axiome}}{\alpha{:}P \Rightarrow Q, \beta{:}P \vdash Q}\ \Rightarrow\text{-élim}$$

peut être exprimée par l'arbre (que nous écrivons comme un terme) $\Rightarrow$-élim(α, β).

• Pour les règles qui modifient le contexte (le $\Rightarrow$-intro et le $\vee$-élim) il faut indiquer les hypothèses mobiles et les nommer. Ainsi, avec le contexte nommé $(\alpha{:}P)$ pour la racine, la preuve

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\alpha{:}P, \beta{:}P \Rightarrow Q \vdash P \Rightarrow Q}\ \text{axiome} \qquad \dfrac{}{\alpha{:}P, \beta{:}P \Rightarrow Q \vdash P}\ \text{axiome}}{\alpha{:}P, \beta{:}P \Rightarrow Q \vdash Q}\ \Rightarrow\text{-élim}}{\alpha{:}P \vdash (P \Rightarrow Q) \Rightarrow Q}\ \Rightarrow\text{-intro}$$

ne peut pas être simplement réécrite sous la forme $\Rightarrow$-intro($\Rightarrow$-élim(β, α)) car, pour pouvoir retrouver le contexte $(\alpha{:}P,\ \beta{:}P \Rightarrow Q)$ du fils de la racine, il faut indiquer que c'est l'hypothèse $P \Rightarrow Q$ nommée β qui est mobile dans cette règle. Nous réécrirons donc l'arbre de preuve comme le terme suivant $\Rightarrow$-intro$(\beta, P \Rightarrow Q, \Rightarrow$-élim$(\beta, \alpha))$.

On observe qu'il y a beaucoup de similarités entre noms d'hypothèses et variables. Par exemple, dans une preuve associée au terme $\Rightarrow$-intro$(\beta, P \Rightarrow Q, \pi)$, le nom d'hypothèse β ne peut être utilisé que dans la sous-preuve associée au terme π et ne figure pas dans le contexte $\alpha{:}P$ de la racine de cette preuve. On peut donc dire que la preuve π est dans le champ de la variable β et que cette variable est liée par le symbole $\Rightarrow$-intro.

Termes associés aux preuves constructives

Nous pouvons maintenant attacher des termes aux preuves en déduction naturelle. Pour chaque règle, nous introduisons un symbole de fonction qui sera le nom de la règle. Par exemple le nom de la règle $\Rightarrow$-intro est λ. L'arité de ce symbole est le nombre de prémisses de la règle plus le nombre de paramètres nécessaires pour retrouver la règle à partir de son schéma. Pour chacun de ses arguments, ce symbole de fonction lie les hypothèses qui sont rajoutées dans la prémisse associée. Formellement, la construction du terme associé à une preuve se fait par la récurrence décrite par la Table 2 (le lecteur qui s'interroge sur les symboles choisis pour construire ces termes trouvera leur raison d'être au §1.6.4).

1.6.3 Réduction de preuve constructive

Coupures

Une *coupure* est une preuve qui se termine par une règle d'élimination dont la prémisse principale est prouvée par une règle d'introduction du même connecteur. Un exemple très simple est celui d'une preuve de la forme suivante :

$$\dfrac{\dfrac{\dfrac{\pi_1 \vdots}{\Gamma \vdash A} \qquad \dfrac{\vdots \pi_2}{\Gamma \vdash B}}{\Gamma \vdash A \wedge B}\ \wedge\text{-intro}}{\Gamma \vdash A}\ \wedge\text{-élim}$$

$\alpha_i \left\{ \dfrac{}{\Gamma \vdash A_i} \text{ axiome} \right.$ $\qquad$ $I \left\{ \dfrac{}{\Gamma \vdash \top} \top\text{-intro} \right.$	$\pi \left\{ \dfrac{\vdots}{\Gamma \vdash \bot} \right. \Rightarrow \delta_\bot(A,\pi) \left\{ \dfrac{\dfrac{\vdots}{\Gamma \vdash \bot}}{\Gamma \vdash A} \bot\text{-élim} \right.$
$\pi_1 \left\{ \dfrac{\vdots}{\Gamma \vdash A} \right., \pi_2 \left\{ \dfrac{\vdots}{\Gamma \vdash B} \right. \Rightarrow \langle \pi_1, \pi_2 \rangle \left\{ \dfrac{\dfrac{\vdots}{\Gamma \vdash A} \quad \dfrac{\vdots}{\Gamma \vdash B}}{\Gamma \vdash A \wedge B} \wedge\text{-i} \right.$	$\pi \left\{ \dfrac{\vdots}{\Gamma \vdash A \wedge B} \right. \Rightarrow \mathit{fst}(\pi) \left\{ \dfrac{\dfrac{\vdots}{\Gamma \vdash A \wedge B}}{\Gamma \vdash A} \wedge\text{-é} \right., \mathit{snd}(\pi) \left\{ \dfrac{\dfrac{\vdots}{\Gamma \vdash A \wedge B}}{\Gamma \vdash B} \wedge\text{-é} \right.$
$\pi \left\{ \dfrac{\vdots}{\Gamma \vdash A} \right. \Rightarrow i(A,B,\pi) \left\{ \dfrac{\dfrac{\vdots}{\Gamma \vdash A}}{\Gamma \vdash A \vee B} \vee\text{-i} \right.$ $\pi \left\{ \dfrac{\vdots}{\Gamma \vdash B} \right. \Rightarrow j(A,B,\pi) \left\{ \dfrac{\dfrac{\vdots}{\Gamma \vdash B}}{\Gamma \vdash A \vee B} \vee\text{-i} \right.$	$\pi_1 \left\{ \dfrac{\vdots}{\Gamma \vdash A \vee B} \right., \pi_2 \left\{ \dfrac{\vdots}{\Gamma, \alpha{:}A \vdash C} \right., \pi_3 \left\{ \dfrac{\vdots}{\Gamma, \beta{:}B \vdash C} \right. \Longrightarrow$ $\delta(\pi_1, \alpha{:}A\ \pi_2, \beta{:}B\ \pi_3) \left\{ \dfrac{\dfrac{\vdots}{\Gamma \vdash A \vee B} \quad \dfrac{\vdots}{\Gamma, \alpha{:}A \vdash C} \quad \dfrac{\vdots}{\Gamma, \beta{:}B \vdash C}}{\Gamma \vdash C} \vee\text{-é} \right.$
$\pi \left\{ \dfrac{\vdots}{\Gamma \vdash A} \right. \Rightarrow \lambda\alpha{:}A\ \pi \left\{ \dfrac{\dfrac{\vdots}{\Gamma, A \vdash B}}{\Gamma \vdash A \Rightarrow B} \Rightarrow\text{-i} \right.$	$\pi_1 \left\{ \dfrac{\vdots}{\Gamma \vdash A \Rightarrow B} \right., \pi_2 \left\{ \dfrac{\vdots}{\Gamma \vdash A} \right.$ $\Longrightarrow \mathit{app}(\pi_1, \pi_2) \left\{ \dfrac{\dfrac{\vdots}{\Gamma, A \vdash B} \quad \dfrac{\vdots}{\Gamma \vdash A}}{\Gamma \vdash B} \Rightarrow\text{-é} \right.$
$\pi \left\{ \dfrac{\vdots}{\Gamma \vdash A} \right. \Rightarrow \lambda x\ \pi \left\{ \dfrac{\dfrac{\vdots}{\Gamma \vdash B}}{\Gamma \vdash \forall x\ A} \forall\text{-i} \right.$	$\pi \left\{ \dfrac{\vdots}{\Gamma \vdash \forall x\ A} \right. \Rightarrow \mathit{app}(\pi, t) \left\{ \dfrac{\dfrac{\vdots}{\Gamma \vdash \forall x\ A}}{\Gamma \vdash A(t/x)} \forall\text{-é} \right.$
$\pi \left\{ \dfrac{\vdots}{\Gamma \vdash A(t/x)} \right. \Rightarrow \langle t, \pi \rangle \left\{ \dfrac{\dfrac{\vdots}{\Gamma \vdash A(t/x)}}{\Gamma \vdash \exists x\ A} \exists\text{-i} \right.$	$\pi_1 \left\{ \dfrac{\vdots}{\Gamma \vdash \exists x\ A} \right., \pi_2 \left\{ \dfrac{\vdots}{\Gamma, \alpha{:}A \vdash B} \right. \Rightarrow$ $\delta_\exists(\pi_1, x\alpha{:}A\ \pi_2) \left\{ \dfrac{\dfrac{\pi_1}{\Gamma \vdash \exists x\ A} \quad \dfrac{\pi_2}{\Gamma, \alpha{:}A \vdash B}}{\Gamma \vdash B} \exists\text{-é} \right.$

TABLE 2 – *Termes associés aux preuves en déduction naturelle constructive*

$$
\begin{array}{rcl}
\mathit{fst}(\langle \pi_1, \pi_2 \rangle) & \longrightarrow & \pi_1 \\
\mathit{snd}(\langle \pi_1, \pi_2 \rangle) & \longrightarrow & \pi_2 \\
\delta(i(A, B, \pi_1), \alpha{:}A\ \pi_2, \beta{:}B\ \pi_3) & \longrightarrow & \pi_2(\pi_1/\alpha) \\
\delta(j(A, B, \pi_1), \alpha{:}A\ \pi_2, \beta{:}B\ \pi_3) & \longrightarrow & \pi_3(\pi_1/\beta) \\
((\lambda\alpha{:}A\ \pi_1)\ \pi_2) & \longrightarrow & \pi_1(\pi_2/\alpha) \\
((\lambda x\ \pi)\ t) & \longrightarrow & \pi(t/x) \\
\delta_\exists(\langle t, \pi_1 \rangle, x\ \alpha{:}A\ \pi_2) & \longrightarrow & \pi_2(t/x, \pi_1/\alpha)
\end{array}
$$

TABLE 3 – *Règles de réduction de preuve constructive*

Une telle preuve peut être simplifiée en ne gardant que la preuve π_1 qui conduit au même séquent $\Gamma \vdash A$. Avec les termes associés, la preuve avec coupure s'écrit $\mathit{fst}(\langle \pi_1, \pi_2 \rangle)$ et la règle de réduction de preuve est

$$\mathit{fst}(\langle \pi_1, \pi_2 \rangle) \longrightarrow \pi_1$$

Un exemple de coupure plus complexe est celui d'une preuve de la forme

$$\dfrac{\dfrac{\dfrac{\vdots \pi_1}{\Gamma, A \vdash B}}{\Gamma \vdash A \Rightarrow B}\ \Rightarrow\text{-intro} \qquad \dfrac{\vdots \pi_2}{\Gamma \vdash A}}{\Gamma \vdash B}\ \Rightarrow\text{-élim}$$

dont le terme associé est $((\lambda\alpha{:}A\ \pi_1)\ \pi_2)$. La réduction de cette preuve se fait ainsi :
• on supprime l'hypothèse A dans tous les séquents de la preuve π_1,
• on remplace chaque axiome qui introduit en conclusion cette formule A par la preuve π_2.
Cette opération est donc une simple substitution : dans la preuve π_1, on substitue la preuve π_2 à chaque occurrence de la variable α associée à l'hypothèse A dans le contexte Γ, A. Ceci conduit à la règle de réduction de preuve

$$((\lambda\alpha{:}A\ \pi_1)\ \pi_2) \longrightarrow \pi_1(\pi_2/\alpha)$$

Les autres règles sont construites de façon analogue. La Table 3 donne l'ensemble des règles de réduction de preuve. Ainsi, si une preuve contient une coupure, il est facile de l'éliminer en appliquant l'une des règles de réduction. Mais l'élimination d'une coupure peut créer d'autres coupures ! Le théorème principal du sujet assure que, néanmoins, *ce processus de réduction des preuves se termine.*

Remarque. La famille des termes ainsi construits avec ses règles de réduction est une extension du λ-calcul évoqué au §1.2.6. Le λ-calcul proprement dit correspond aux termes associés aux preuves dans lesquelles $\Rightarrow$ est le seul connecteur qui apparaisse avec la seule règle afférente, dite β-réduction, qui est celle de la cinquième ligne de la Table 3.

Elimination des coupures et extraction du témoin

Ce mécanisme de réduction des preuves est exactement celui qui permet d'extraire le témoin des preuves (cf. §1.6.1). En effet, une récurrence simple sur la structure des preuves montre que si une preuve constructive d'un séquent sans axiome (c'est-à-dire d'un séquent de contexte vide) est sans coupure alors cette preuve se termine par une règle d'introduction.
En particulier, si π est une preuve d'une formule de la forme $\exists y\ A$ alors π se termine par une règle $\exists$-intro et elle est donc de la forme $\langle u, \pi' \rangle$, où π' est une de la formule $A(u/y)$. Ainsi, le terme u est le témoin cherché.

1.6.4 L'interprétation de Brouwer-Heyting-Kolmogorov

La propriété du témoin fait du langage des preuves un langage de programmation. Ce qui frappe le plus dans ce langage, c'est sa similarité avec les langages de programmation fonctionnelle bien connus. Cette similarité ne s'explique pas par la propriété du témoin mais par une observation toute différente, due à Brouwer, Heyting et Kolmogorov.

Pour construire une preuve de $A \wedge B$ avec la règle $\wedge$-intro, il faut d'abord construire une preuve de A et une preuve de B. Et quand on a une preuve de $A \wedge B$, on peut l'utiliser avec les règles $\wedge$-élim pour construire une preuve de A ou une preuve de B. Ceci rappelle ce qui se passe en théorie des ensembles avec le couple. Pour construire un couple formé d'une preuve de A et d'une preuve de B, il faut d'abord construire une preuve de A et une preuve de B. Et quand on a cette un couple formé d'une preuve de A et d'une preuve de B, on peut l'utiliser pour avoir une preuve de A ou une preuve de B. Ainsi, une preuve de $A \wedge B$ se construit et s'utilise exactement comme un couple formé d'une preuve de A et d'une preuve de B.

De la même façon, une preuve de $A \Rightarrow B$ se construit et s'utilise exactement comme un algorithme qui associe à une preuve de A une preuve de B.

Ainsi, une preuve de $A \wedge B$ *est* un couple formé d'une preuve de A et d'une preuve de B, et une preuve de $A \Rightarrow B$ est un algorithme qui associe à une preuve de A une preuve de B. Plus généralement,

- Une preuve de $\top$ est l'élément d'un ensemble singleton $\{I\}$.
- Il n'existe pas de preuve de $\bot$.
- Une preuve de $A \wedge B$ est un couple formée d'une preuve de A et d'une preuve de B.
- Une preuve de $A \vee B$ est ou bien une preuve de A ou bien une preuve de B.
- Une preuve de $A \Rightarrow B$ est un algorithme qui associe à une preuve de A une preuve de B.
- Une preuve de $\forall x\ A$ est un algorithme qui associe à chaque terme t une preuve de $A(t/x)$.
- Une preuve de $\exists x\ A$ est un couple formée d'un terme t et d'une preuve de $A(t/x)$.

Ce sont ces correspondances qui justifient les noms choisis pour les symboles entrant dans les termes associés aux preuves. Si π_1 est une preuve de A et π_2 une preuve de B, nous écrivons $\langle \pi_1, \pi_2 \rangle$ pour la preuve de $A \wedge B$ construite à l'aide de la règle $\wedge$-intro parce que cette preuve est le couple formé d'une preuve de A et d'une preuve de B.
Si π est une preuve de B qui utilise une hypothèse A représentée par une variable libre α, nous écrivons $\lambda\alpha{:}A\ \pi$ pour la preuve de $A \Rightarrow B$ construite à l'aide de la règle $\Rightarrow$-intro parce que cette preuve est un algorithme qui associe à une preuve de A une preuve de B.

De plus, comme l'ont remarqué Curry, de Bruijn et Howard, on peut associer un type à la

famille des preuves de chaque formule. En particulier, si $\Phi(A)$ est le type des preuves de la formule A et $\Phi(B)$ celui de la famille des preuves de la formule B alors le type de la famille des preuves de la formule $A \Rightarrow B$ n'est autre que $\Phi(A) \rightarrow \Phi(B)$, c'est-à-dire le type des fonctions de $\Phi(A)$ dans $\Phi(B)$, puisque toute preuve de $A \Rightarrow B$ est un algorithme envoyant une preuve de A sur une preuve de B. Ainsi,

$$\Phi(A \Rightarrow B) = \Phi(A) \rightarrow \Phi(B)$$

Des phénomènes similaires sont aussi vrais avec les autres connecteurs et quantifications : Φ est un isomorphisme entre les formules et les types. C'est l'*isomorphisme de Curry-de Bruijn-Howard*. En identifiant, comme il est usuel de le faire, des objets isomorphes, on voit que le type d'une preuve est la formule qu'elle prouve.

1.6.5 Théories

Rappelons que lorsqu'on a esquissé la démonstration de la propriété du témoin au §1.6.3, on a bien précisé qu'une preuve constructive sans coupure d'un séquent sans axiome se termine nécessairement par une règle d'introduction. Cette propriété ne s'étend pas à n'importe quelle théorie axiomatique. Par exemple, si on ajoute le seul axiome $\exists x\ P(x)$, alors la formule $\exists x\ P(x)$ admet une preuve qui se réduit à cet axiome et ne se termine donc pas par une règle d'introduction. Et, bien que la formule $\exists x\ P(x)$ soit prouvable, il n'existe aucun terme t tel que la formule $P(t)$ soit prouvable.

Evidemment, lorsqu'on n'a aucun axiome, il y a bien peu de fonctions dont on peut prouver l'existence et le langage des preuves constructives est alors un langage de programmation assez pauvre. C'est donc une des questions les plus essentielles du sujet de l'interprétation algorithmique des preuves que d'essayer d'incorporer des axiomes sans perdre la propriété capitale qu'une preuve constructive sans coupure se termine nécessairement par une règle d'introduction.

Davantage de règles de réduction

La première façon d'incorporer des axiomes est de rajouter au langage une constante pour chacun d'eux. Par exemple, on peut rajouter une constante Rec^A pour chaque instanciation du schéma de récurrence. Si π_1 est une preuve de $A(0/x)$ et π_2 une preuve de $\forall y\ (A(y/x) \Rightarrow A(S(y)/x))$, alors le terme $(Rec^A\ \pi_1\ \pi_2)$ représentera une preuve de $\forall x\ A$, et, pour chaque entier naturel n, le terme $(Rec^A\ \pi_1\ \pi_2\ n)$ représentera une preuve de $A(n/x)$. Si π_1 et π_2 sont sans coupure, alors cette preuve sera aussi sans coupure. Cependant, cette preuve se termine par une règle $\forall$-élim rule et non pas une règle d'introduction. Aussi, afin de retrouver la propriété du témoin, nous devrons rajouter de nouvelles règles de réduction, par exemple les règles

$$\begin{array}{rcl} (Rec^A\ \pi_1\ \pi_2\ 0) & \longrightarrow & \pi_1 \\ (Rec^A\ \pi_1\ \pi_2\ S(n)) & \longrightarrow & (\pi_2\ n\ (Rec^A\ \pi_1\ \pi_2\ n)) \end{array}$$

Cette approche a été beaucoup étudiée, citons K. Gödel (98), W.W. Tait (221), P. Martin-Löf (157), Ch. Paulin (178), B. Werner (235), ...

Davantage de règles de déduction

Une alternative à la solution précédente est d'abandonner les axiomes et de les remplacer par des règles de déduction non logiques. Par exemple, on pourra abandonner l'axiome de l'ensemble des parties

$$\forall x \forall y \; (x \in \mathcal{P}(y) \Leftrightarrow x \subseteq y)$$

et le remplacer par deux règles :

$$\frac{x \subseteq y}{x \in \mathcal{P}(y)} \text{ fold} \qquad , \qquad \frac{x \in \mathcal{P}(y)}{x \subseteq y} \text{ unfold}$$

Dans ce cas, il faut bien sûr rajouter aussi des règles de réduction de preuves. En particulier, l'application successive des règles *fold* et *unfold* doit être considérée comme une coupure et la preuve

$$\dfrac{\dfrac{\dfrac{\pi}{x \subseteq y}}{x \in \mathcal{P}(y)} \text{ fold}}{x \subseteq y} \text{ unfold}$$

doit pouvoir être réduite à π. Cette approche a également été beaucoup étudiée, citons D. Prawitz (184), M. Crabbé (60; 61), L. Hallnäs (114), J. Ekman (80), S. Negri and J. von Plato (168), B. Wack (233), ...

Davantage de formules

Certains axiomes peuvent être abandonnés si on enrichit le langage des formules. Un exemple typique est celui des axiomes de la théorie simple des types : ils peuvent abandonnés si l'on permet de quantifier sur les prédicats. Ceci conduit à étendre donc le langage des types. C'est une approche qui a été étudiée par J.-Y. Girard (94), Th. Coquand & G. Huet (57), D. Leivant (142), J.-L. Krivine and M. Parigot (138), ...

Des règles de réécriture

Enfin, on peut aussi abandonner des axiomes si on permet d'identifier certains termes et/ou certaines formules. Par exemple, on peut abandonner l'axiome $\forall x \; (x + 0 = x)$ si l'on identifie les termes $x + 0$ et x, on peut aussi abandonner l'axiome de récurrence si l'on identifie les formules $N(y)$ et $\forall c \; ((0 \; \epsilon \; c) \Rightarrow \forall x \; (N(x) \Rightarrow x \; \epsilon \; c \Rightarrow S(x) \; \epsilon \; c) \Rightarrow y \; \epsilon \; c)$.

Cette approche, qui vise à synthétiser et aller au-delà des trois autres approches mentionnées plus haut, a été étudiée dans (73; 74; 59), ...

Dans toutes ces approches, la propriété de terminaison des réductions est conservée pour certaines théories mais perdue pour d'autres. Lorsqu'elle est conservée, on obtient alors un langage de programmation dont tous les programmes terminent et dans lequel on peut programmer toutes les fonctions dont l'existence est prouvable dans la théorie considérée.

1.6.6 Autres extensions

A part le problème de l'incorporation d'axiomes, une autre question très importante est celle d'étendre l'interprétation algorithmique des preuves à la logique classique (c'est la lo-

gique obtenue en rajoutant n'importe laquelle des trois formules de la Remarque à la fin du §1.6.2). Ici, la propriété du témoin doit être remplacée par le théorème de Herbrand :

> *D'une preuve constructive d'une formule de la forme $\exists y\ A$, il est possible d'extraire une suite de termes $u_1, ..., u_n$ et une preuve de la formule*
>
> $$A(u_1/y) \vee ... \vee A(u_n/y)$$

De cette façon, *les preuves classiques apparaissent comme des algorithmes non déterministes.* Voir, par exemple, (175).

Enfin, à part la déduction naturelle, d'autres calculs logiques ont été considérés, en particulier le calcul des séquents. Voir, par exemple, (62; 226).

1.6.7 Elimination des coupures et résultats de cohérence

Bien avant l'isomorphisme de Curry-Howard, un lien remarquable entre déduction et calculabilité était apparu avec l'un des résultats fondamentaux les plus inattendus du vingtième siècle, à savoir le théorème d'incomplétude de Gödel, 1931 (97), dont la preuve est basée sur une notion (informelle) de déduction algorithmique. Ce théorème affirme qu'aucun système de preuve ne peut capturer pleinement le raisonnement mathématique : toute théorie suffisante pour capturer les raisonnements arithmétiques est nécessairement incomplète, c'est-à-dire telle qu'il existe des énoncés qui ne sont pas démontrables et dont la négation n'est pas non plus démontrable. En particulier, on peut exprimer la cohérence (c'est-à-dire la non contradiction) d'une théorie mathématique par un énoncé, qui ne peut être ni démontré ni infirmé (du moins si la théorie est cohérente).

Avant que ce théorème n'apparaisse, le mathématicien David Hilbert avait invité la communauté logique à assurer la bonne fondation des mathématiques en prouvant la cohérence des théories mathématiques par des moyens "finitistes", donc formalisables dans ces théories. C'est ce qui est appelé le *programme de Hilbert.* Bien sûr, le théorème de Gödel a obligé à une révision drastique de ce programme. Mais pas à son abandon : il fallait trouver des méthodes capables de surmonter cette impossibilité – donc nécessairement non formalisables dans la théorie elle-même – qui restent néanmoins "convainquantes". Le premier à y réussir a été Gerhard Gentzen (un élève de Hilbert) : il a prouvé la cohérence de l'arithmétique de Peano à l'aide d'une induction transfinie (non formalisable dans cette théorie), 1936 (93). L'argument de Gentzen a été plus tard transformée par Kurt Schütte, 1951 (197) (autre élève de Hilbert), en une démonstration de l'élimination des coupures dans une formalisation de l'arithmétique où (selon la stratégie du §1.6.5) les axiomes de récurrence sont remplacés par une règle avec une infinité de prémisses, dite ω-règle :

$$\frac{A(0), A(1), A(2), \ldots}{\forall x\ A(x)}\ \omega$$

Cette propriété d'élimination des coupures implique assez facilement la cohérence de la théorie : si la théorie était incohérente, on pourrait y prouver $\perp$, donc aussi prouver $\perp$ par une preuve sans coupure, laquelle doit nécessairement se terminer par une introduction, qui ne peut évidemment être qu'une introduction de $\perp$ (puisque la formule prouvée se réduit à $\perp$), mais il n'y a pas de telle règle...

1.7 Décidable versus indécidable

Quand on ne retient que la possibilité de calcul et non les manières de le faire

1.7.1 Démonstration automatique : décidabilité et théories logiques

> "*Non disputemus, sed calculemus*".
> La seule façon de rectifier nos raisonnements est de les rendre aussi tangibles que ceux des mathématiciens, de sorte qu'une erreur puisse se voir d'un coup d'oeil. Et lorsqu'il y aura une dispute entre deux personnes, on pourra simplement dire : "Ne discutons pas, Monsieur, calculons pour voir qui a raison".
>
> Gottfried Wilhelm Leibniz, 1687

Comme on le voit, l'idée de la démonstration automatique (avec une ambition considérable) remonte à Leibniz (1646-1716), philosophe, mathématicien et touche à tout de génie. Pour parvenir à un tel but, il se proposait aussi de trouver une langue universelle capable d'exprimer toutes les idées possibles : la "lingua characteristica universalis". De nos jours, l'ambition s'est drastiquement réduite. On ne cherche plus de langue universelle : on considère une multitude le langages logiques adaptés à des structures mathématiques bien définies. Pour chacun de ces langages, on se demande si la famille des énoncés vrais est décidable, c'est-à-dire forme un ensemble dont on peut décider par un algorithme si un énoncé donné y figure ou pas. Nous donnons ci-dessous quelques exemples de l'incroyable foison de résultats obtenus depuis bientôt un siècle.

Entscheidungsproblem (le problème de la décision)

Etant donné un langage logique, on sait qu'un énoncé est vrai dans toutes les structures de ce langage si et seulement s'il est prouvable dans le calcul des prédicats de ce langage (c'est le théorème de complétude de Gödel). Il est alors intéressant de savoir si cet ensemble de théorèmes est ou non une ensemble calculable d'énoncés : c'est le problème de la décision. La réponse à ce problème (pour les langages avec égalité) est la suivante.

- Décidable si le langage ne contient que des symboles de relations unaires (outre l'égalité) et des constantes et au plus une fonction unaire (Rabin, 1969).
- Indécidable sinon, c'est-à-dire dès que le langage contient au moins une relation (autre que l'égalité) ou une fonction binaire ou ternaire ou... ou bien au moins deux fonction unaires.

Dans le cas où la réponse est négative, on peut rafiner ce problème et demander s'il y a des classes intéressantes de formules pour lesquelles ce problème devient décidable. La réponse dépend alors, d'une part du langage considéré, d'autre part de la classe considérée. La réponse est aujourd'hui complète et plutôt technique, mais elle a mis près de trois-quarts de siècle à être apportée ! C'est ainsi que les classes préfixielles décidables des langages avec égalité sont les suivantes :

- Formules de la forme $\forall \ldots \forall$ quel que soit le langage (Gurevich, 1976).
- Formules de la forme $\forall \ldots \forall \exists \ldots \exists$ si le langage est purement relationnel, c'est-à-dire ne contient aucun symbole de fonction (Ramsey, 1930).
- Formules de la forme $\forall \ldots \forall \exists \forall \ldots \forall$ si le langage ne contient que des relations et au plus une fonction unaire (Shelah, 1977).

Arithmétique des entiers

Arithmétique avec addition et multiplication. Maintenant, nous considérons la famille des énoncés vrais dans l'arithmétique des entiers : $\mathbb{N}$ avec addition et multiplication usuelles. L'indécidabilité de l'arithmétique est un corollaire assez trivial du théorème d'incomplétude de Gödel qui a été explicité par Church en 1935.

Indécidabilité de l'arithmétique diophantienne. Le meilleur (ou pire) résultat connu à ce jour est le "problème diophantien" suivant (Yuri Matijasevitch & Julia Robinson & Martin Davis) :

> *Il existe deux polynômes P, Q (à coefficients dans $\mathbb{N}$) fixés tels que l'ensemble des entiers a pour lesquels l'énoncé*
>
> $$\exists x_1 \ldots \exists x_m \; P(a, x_1, \ldots, x_m) = Q(a, x_1, \ldots, x_m)$$
>
> *est vrai n'est pas calculable.*

On peut même supposer que $m = 9$ (Matijasevich (158), cf. Jones (126)). On soupçonne que l'on peut, en fait, avoir $m = 3$ mais on sait que l'on ne peut pas avoir $m = 2$.

Arithmétique purement additive. On renonce à la multiplication. Cette arithmétique additive est alors décidable (Presburger, 1929). Ce résultat est d'une très grande importance en informatique car elle permet de modéliser nombre de situations et fournit donc des algorithmes.

Arithmétique du successeur avec quantifications sur les entiers et aussi sur les ensembles d'entiers. Cette arithmétique est également décidable (Büchi, 1959). La preuve s'obtient par des techniques d'automates finis. Là encore, ce résultat est d'une importance très grande en informatique.

Arithmétique des nombres rationnels

Elle est également indécidable. En fait, il est possible de définir l'ensemble des entiers dans la structure algébrique des rationnels. Il s'agit d'un résultat difficile de Julia Robinson, 1948 (187), qui est la source d'une myriade d'autres résultats d'indécidabilité.

Théorie des mots

Mots et concaténation. On considère les mots d'un alphabet Σ ayant au moins deux lettres. L'opération de concaténation est celle qui met deux mots bout à bout. La théorie en est (sans surprise) indécidable (Quine, 1946) car il est facile d'y coder l'arithmétique avec addition et multiplication. La surprise est la décidabilité (Makanin, 1979) de la classe des énoncés de la forme

$$\exists x_1 \ldots \exists x_k \; C(x_1, \ldots, x_k) = D(x_1, \ldots, x_k)$$

où C, D sont des mots construits avec l'alphabet Σ augmenté des symboles de variables (qui représentent des mots, pas des lettres).

Mots et adjonction d'une lettre à la fin. On remplace la concaténation par des opérations unaires ; si $a \in \Sigma$, l'opération $Succ_a$ consiste à rajouter la lettre a en fin du mot. Cette théorie est décidable même si on s'autorise à quantifier à la fois sur les mots et sur les ensembles de mots (Rabin, 1969). Il s'agit d'un résultat très difficile qui est l'un des fleurons de l'informatique théorique.

Algèbre réelle

Si l'arithmétique des entiers et celle des rationnels est indécidable (cf. plus haut), celle des réels $(\mathbb{R}, +, \times, =)$ est - ô surprise - décidable (Tarski, 1931 (222)). Tous les problèmes de solutions d'équations de degré quelconque posés au lycée sont donc résolubles à l'aide d'une machine... Avouons quand même que la complexité de cette théorie est fort grande, en espace exponentiel.
En revanche, on ignore si la théorie de $(\mathbb{R}, +, \times, x \mapsto e^x, =)$ (on a rajouté la fonction exponentielle) est ou non décidable. On sait (152) que le problème est lié à une question très difficile de théorie des nombres, la conjecture de Schanuel (1960).

Géométrie élémentaire

Les anciens qui ont passé de durs moments sur les problèmes de géométrie du plan ou de l'espace avec les côniques et les quadriques, le cercle des neuf points, les faisceaux harmoniques, les problèmes de géométrie descriptive,... trouveront une saveur particulière à ce résultat de Tarski, 1931 : la théorie de la géométrie élémentaire est décidable (son axiomatisation se fait avec des notions de points, droites, plans, une appartenance et une relation "être entre deux points"). Ainsi, tous les problèmes du bachot d'il y a bien des années pouvaient être résolus par une machine... Ce résultat est, en fait, un corollaire facile de la décidabilité de l'algèbre réelle qui utilise le passage en coordonnées cartésiennes.

1.7.2 Quelques autres problèmes

Dominos. Supposons donnée une famille finie de carrés ayant tous la même taille et dont les cotés sont coloriés de diverses façons. *Peut-on paver le plan euclidien $\mathbb{Z} \times \mathbb{Z}$ avec de tels carrés de sorte que si deux carrés sont adjacents alors leur coté commun est colorié de la même façon dans ces deux carrés ?* Ce problème est indécidable : l'ensemble des familles finies pour lesquelles on peut paver ainsi n'est pas récursif (Berger, 1966 (14)).
On peut poser le même problème dans le plan hyperbolique. Attention, il n'y a pas de carré (ni de rectangle) dans le plan hyperbolique mais, en revanche, pour tout $s \geq 5$, il y a des polygones réguliers à angles droits ayant s cotés... Pour chaque $s \geq 5$, la réponse est la même, le problème est indécidable (Margenstern, 2007 (154; 155)).

Réseaux de Petri. Un modèle particulièrement fructueux de la concurrence est celui des réseaux de Petri (Carl Adam Petri, 1962). Une forme simple de tel réseau est constitué des éléments suivants :
- deux familles finies fixées disjointes S, T : les "places" et les "transitions",
- une famille finie fixée d'arcs $F \subseteq (S \times T) \cup (T \times S)$ entre places et transitions,
- une distribution dynamique de "jetons" $M : S \to \mathbb{N}$ sur les places.

Les entrées (resp. sorties) d'une transition $t \in T$ sont toutes les places s telles que $(s, t) \in F$ (resp. $(t, s) \in F$), c'est-à-dire telles qu'il y a un arc de s vers t (resp. de t vers s).
Une distribution M admet un successeur s'il existe une transition $t \in T$ telle que $M(s, t) > 0$ pour toute entrée s de t. Le successeur de la distribution M relativement à une telle transition t est la distribution M' obtenue en retranchant un jeton à chaque entrée de t puis en rajoutant un jeton à chaque sortie de t.
Le fonctionnement d'un réseau de Petri est alors une suite de distributions successives (noter

que ceci n'est pas déterministe, il peut y avoir plusieurs telles suites possibles). Une telle suite peut être bloquée s'il n'y a plus de successeur possible. Le problème le plus difficile du sujet est celui de l'accessibilité : peut-on atteindre telle distribution à partir d'une distribution donnée ? Ce problème est décidable (en espace exponentiel), il s'agit d'un résultat très difficile et profond (Mayr, 1981 (159)).

1.7.3 Décider avec une bonne probabilité de réponse exacte

Considérons une machine de Turing non déterministe : il peut y avoir plusieurs transitions possibles (mais un nombre fini, bien sûr) à certaines étapes du calcul. Supposons que l'on mette une distribution de probabilités sur ces différentes transitions possibles. On peut alors considérer l'ensemble des entrées pour lesquels la probabilité d'être accepté est $\geq \delta$ pour un certain $\delta > 0$ fixé. Un argument par simulation en parallèle de tous les calculs possibles permet de voir que cet ensemble est, en fait, récursivement énumérable, c'est-à-dire reconnaissable par une machine de Turing déterministe, cf. (141), 1956. Ainsi, dans le cas de la reconnaissance de langage, les probabilités n'apportent rien de plus.

1.8 A l'intérieur de la calculabilité : la théorie de la complexité

1.8.1 Limites des ressources physiques

Temps des Dieux, temps des hommes. Sur l'île de Calypso, Ulysse renonce à ce paradis miniature, à l'éternité pour reprendre son identité d'homme adulte qui vieillit et meurt. La mythologie grecque enseigne qu'il y a au moins deux sortes de temps. Celui des dieux et de l'éternité où tout est immobile. Celui des hommes qui est un temps linéaire. La calculabilité s'intéresse à savoir si un problème est décidable sans relation au temps, où en quelque sort tout est déjà là. Ce rapport éternel aux temps correspond à la différence entre la calculabilité et la complexité. Prendre le chemin de la théorie de la complexité, c'est descendre de l'Olympe et réintégrer le temps linéaire comme une donnée fondamentale à la notion de calcul.

Borne de temps (selon la physique) : 10^{41}. La durée la plus grande qui ait un sens physique est l'âge A de l'univers qui est de l'ordre de 15×10^9 années, soit $15 \times 10^9 \times 365.25 \times 24 \times 3600 \sim 4,734 \times 10^{17}$ secondes (ceci dans le cadre du modèle cosmologique du Big Bang). La durée la plus courte τ qui ait un sens physique est le temps que met un photon pour parcourir, à la vitesse de la lumière $c = 3 \times 10^8$ m/s, une distance égale au diamètre $d = 10^{-15}$ m du proton. Soit $\tau = \frac{d}{c} = \frac{10^{-15}}{3 \times 10^8} \sim 3,333 \times 10^{-24}$ seconde.
Le nombre $N = \frac{A}{\tau} = \frac{4,734 \times 10^{17}}{3,333 \times 10^{-24}} \sim 10^{41}$ représente alors le plus grand nombre d'unités de temps ayant un sens physique.

Borne de matière (selon la physique) : $\leq 10^{80}$ particules. Dans le fil de la réflexion précédente, il est raisonnable de considérer aussi les limites de notre univers : il semble que le nombre de particules élémentaires soit borné par $10^{80} (\leq 2^{320})$.

La théorie de la complexité comme théorie de la calculabilité "raisonnable". Les chiffres

que l'on vient de voir montrent qu'un algorithme en temps ou espace exponentiel a une complexité rédhibitoire. La théorie de la complexité étudie les problèmes qui peuvent être résolus avec des ressources de calcul limitées. Citons quelques livres ouvrages de référence : le classique de Papadimitriou (174), le livre de Jones (127) orienté théorie de la programmation, le livre de Savage (193) orienté algorithmique et calcul avec différents modèles de calcul, ainsi que les livres (213; 99; 5).

1.8.2 Complexité de quelques problèmes

Exemple 1 : Multiplication de deux entiers. Un même problème peut être résolu par plusieurs algorithmes très différents. Prenons l'exemple de la multiplication de deux entiers a et b ayant respectivement p et q chiffres. L'algorithme appris à l'école calcule en effectuant $O(p.q)$ opérations. Posant $n = \max(p, q)$, on dira que la complexité de cet algorithme est en $O(n^2)$. Il y a d'autres algorithmes plus rapides mais, conceptuellement, plus difficiles (on échange donc du temps contre de la matière grise...). Celui de Karatsuba, 1962 (129), est en $O(n^{\log_2(3)})$, il consiste à couper la représentation binaire des entiers en deux : supposons pour simplifier que $p = q = n = 2m$ et écrivons $a = 2^m a' + a''$ et $b = 2^m b' + b''$ où a', a'', b', b'' sont des entiers ayant m chiffres binaires, alors

$$\begin{aligned} a \times b &= (2^m a' + a'') \times (2^m b' + b'') = a'b'\, 2^{2m} + (a'b'' + a''b')\, 2^m + a''b'' \quad (1.1) \\ &= a'b'\, 2^{2m} + [(a' + a'')(b' + b'') - a'a'' - b'b'']\, 2^m + a''b'' \quad (1.2) \end{aligned}$$

Le passage de la ligne (1) à la ligne (2), a priori bizarre, permet de ramener quatre multiplication de deux entiers de m chiffres (à savoir, $a'b'$, ab'', $a''b'$ et $a''b''$) à trois multiplications d'entiers de m (ou $m + 1$) chiffres, (à savoir $a'a''$, $b'b''$ et $(a' + a'')(b' + b'')$). C'est cette astuce qui permet, quand on l'itère, d'obtenir un algorithme en temps $O(n^{\log_2(3)})$ (noter que $n^2 = n^{\log_2(4)}$). Un autre algorithme, celui de Schönhage & Strassen (196), basé sur la transformée de Fourier rapide, est en $O(n \log(n) \log(\log(n)))$. Le meilleur algorithme connu à ce jour est dû à Martin Fürer, 2007 (87), il est en $O(n \log(n) \log^*(n))$ donc quasi-linéaire [6] . Mentionnons enfin que si l'algorithme de Karatsuba est implémenté dans tous les systèmes de calcul formel, ceux de Schönhage & Strassen et celui de Fürer ne le sont pas car, actuellement, ils sont grevés par une constante multiplicative gigantesque (constante cachée dans la notation $O(\ldots)$).

Exemple 2 : Evaluation d'un polynôme. La complexité d'un problème peut aussi dépendre des conditions d'utilisation. Prenons l'exemple de l'évaluation d'un polynôme de degré 4, soit $P(x) = a_4 x^4 + \ldots + a_0$. L'algorithme de Horner, 1819 (121), réécrit ce polynôme comme suit pour l'évaluer à l'aide de 4 multiplications et 4 additions :

$$P(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + x a_4)))$$

On peut montrer que c'est optimum tant en additions (Alexander Ostrowski, 1954) qu'en multiplications (Victor Pan, 1966 (173)). Mais, pour évaluer ce polynôme en un grand nombre

6. La fonction $\log^*$ est le nombre de fois qu'il faut itérer le logarithme pour trouver 0 (i.e. $\log^* 1 = 1$ et $2^p \leq n < 2^{p+1} \Rightarrow \log^* n = 1 + \log^* p$). Bien que cette fonction $\log^*$ tende vers l'infini, sa croissance est d'une lenteur extrême : ainsi, $\log^*(10^{320}) = 5$. Au-regard des limites de temps vues en §1.8.1, cette fonction $\log^*$ ne dépassera donc jamais, en pratique, la valeur $5\ldots$

N de points (par exemple, pour en tracer le graphe), la meilleure complexité n'est pas de 4N additions et 4N multiplications. Un préconditionnement permet de le faire avec environ $3N$ multiplications et $5N$ additions (on a donc échangé environ N multiplications contre autant d'additions). Le "environ" traduit les quelques opérations arithmétiques du calcul des coefficients $\alpha, \beta, \gamma, \delta$ ci-dessous : on peut écrire

$$\begin{aligned} P(x) &= a_4[(x+\alpha)(x+\beta)]^2 + \gamma(x+\alpha)(x+\beta) + (x+\delta) && (1.3)\\ &= (x+\delta) + (x+\alpha)(x+\beta)[\gamma + a_4(x+\alpha)(x+\beta)] && (1.4) \end{aligned}$$

où $\alpha, \beta, \gamma, \delta$ sont tels que $\begin{cases} 2a_4(\alpha+\beta) &= a_3 \\ a_4(\alpha^2+\beta^2+4\alpha\beta)+\gamma &= a_2 \\ 2a_4\alpha\beta(\alpha+\beta)+\gamma(\alpha+\beta)+1 &= a_1 \\ a_4\alpha^2\beta^2+\alpha\beta\gamma+\delta &= a_0 \end{cases}$

La première équation donne la valeur de $\alpha+\beta$. De la deuxième on déduit $\gamma = A + B\alpha\beta$ (où A, B s'expriment avec $a_0, \ldots, a_4$). La troisième donne alors la valeur de $\alpha\beta$. Ayant leur somme et leur produit, on en déduit les valeurs de α et β. La dernière équation donne la valeur de δ. Enfin, la formule (1.4) demande 5 additions et 3 multiplications.
Plus généralement, on peut montrer, qu'avec un préconditionnement, il suffit de n additions et $\lfloor \frac{n}{2} \rfloor + 2$ multiplications pour évaluer un polynôme de degré n (cf. le livre de Knuth (135), p.471-475). Mentionnons quelques résultats spectaculaires de complexité.
Exemple 3 : FFT. Peut-être le plus important résultat est l'algorithme de transformation de Fourier (discrète) rapide (FFT) en temps $O(n \log n)$ (alors qu'a priori, la complexité attendue était en $O(n^2)$) Il s'agit d'une technique difficile mais d'une incroyable utilité. Sans elle, codage et décodage des images serait beaucoup trop longs : c'est la FFT qui nous a permis d'avoir les images de Mars, Jupiter,.... N'hésitons pas à le dire, la FFT est une révolution technologique essentielle (et complètement méconnue).

Exemple 4 : Plus court chemin dans un graphe. Pour trouver le plus court chemin d'un point donné à n'importe quel autre dans un graphe, un temps en $O(n \log n)$ est suffisant. C'est l'algorithme de Dijkstra .

Exemple 5 : Diagrammes de Voronoï. Là encore, il y a un algorithme en $O(n \log n)$.

1.8.3 Complexité d'un problème

Les "Turing Award Lectures" de Rabin (186) et de Cook (50) sont deux articles remarquables sur la génèse de la théorie de la complexité que nous recommandons au lecteur.

Peut-on définir la complexité d'un problème ? Parfois. Une façon de définir la complexité d'un problème est d'établir une borne inférieure et une borne supérieure. Si elles se correspondent de près, on peut considérer qu'on a obtenu la complexité exacte du problème considéré. Un exemple simple est celui des tris. Il y a $n! = n(n+1)\ldots 1$ permutations de n objets distincts. Chaque comparaison entre deux objets divise par deux cette famille de permutations. Pour trier n objets (par des comparaisons successives), il faudra donc au moins $\log(n!)$ comparaisons. Comme $\log(n!)$ est de l'ordre de $n\log(n)$, on obtient une borne inférieure de de l'ordre de $n\log(n)$ comparaisons. Par ailleurs, on connaît des tris en $O(n \log n)$ comparaisons, comme le tri fusion ("merge sort"). La complexité du tri est donc de l'ordre de $n \log n$.

Pas toujours de meilleur algorithme : le théorème d'accélération de Blum. Dans une première approche naïve, on peut penser que, pour tout problème algorithmique, il y a un meilleur algorithme. Mais les choses sont plus compliquées que cela. Citons un résultat à méditer.
Théorème (Manuel Blum, 1967 (24)). *Quelle que soit la fonction calculable croissante* $\alpha : \mathbb{N} \to \mathbb{N}$ *qui tend vers* $+\infty$ *(intuition :* α *croît très lentement, donc* $\alpha(p) \ll p$*), il existe une fonction calculable* $f : \mathbb{N} \to \mathbb{N}$ *qui est calculable et telle que, quel que soit l'algorithme* $\mathcal{A}$ *qui calcule* f*, si, quel que soit* n*,* $\mathcal{A}$ *travaille en temps (resp. en espace)* $C(n)$ *sur l'entrée* n*, alors il existe un autre algorithme* $\mathcal{B}$ *qui calcule* f *et, quel que soit* n*, travaille en temps (resp. en espace)* $\alpha(C(n))$ *sur l'entrée* n*.*
Une telle fonction f n'admet donc pas de meilleur algorithme puisque chacun de ses algorithmes peut être accéléré.

Complexité et modèle de calcul. La complexité d'un problème peut fortement dépendre du modèle de calcul considéré. C'est ainsi que la reconnaissance des palindromes (les mots qui coïncident avec leur renversé) peut se faire de façon évidente en temps linéaire sur une machine de Turing à deux rubans mais exige $O(n^2)$ étapes de calcul sur une machine de Turing à un seul ruban (un résultat qui peut être établi par un élégant argument de complexité de Kolmogorov (176)). Fort heureusement, les modèles raisonnables de calcul séquentiel (ceux vus en §1.2.2) sont robustes, dans le sens où un modèle peut en simuler un autre avec une perte de temps polynômiale (même quadratique, le plus souvent). Certains auteurs ((214), la monographie (5)) expriment cette robustesse sous la forme d'une *variante forte de la thèse de Church* présentée au §1.2.7 :

> *Tout modèle de calcul physiquement réalisable peut être simulé par une machine de Turing au prix d'un surcoût polynomial.*

De telles versions sont sujettes à plus de controverses que les autres variantes. Par exemple, il n'est pas facile de parler de complexité en λ-calcul. Et, dans le cadre de la comparaison entre machines analogiques et machines digitales, la preuve de cette thèse de Church forte pour les systèmes dynamiques à temps continu présentée dans (230) se fait au prix de simplifications discutables.

Complexité en temps, complexité en espace. Les ressources les plus courantes sont le temps et l'espace dans le cas des algorithmes séquentiels, voir Cook (50) Mais comment mesurer ces ressources ? *Qu'est-ce qu'une étape de calcul ? Qu'est-ce qu'une unité d'espace de calcul ?* Ceci nous renvoie à la notion d'algorithme vue au §1.4 et à la réflexion de Gurevich. Voir aussi Cobham (45). Nous dirons que le temps est le nombre d'étapes qu'effectue une machine pour exécuter un algorithme, et l'espace est la quantité de mémoire nécessaire. Examinons cela de près. Dans l'exemple vu plus haut de la multiplication de deux entiers, est-ce raisonnable de considérer que le coût de l'addition de deux entiers vaut un. Cook a proposé de charger un coût logarithmique (52) dans le cas des RAM, voir 1.2.3.
De même pour les algorithmes parallèles, la complexité sera mesurée par rapport à un modèle de calcul parallèle comme les PRAM (parallel RAM) ou les circuits booléens. Suivant les modèles de calcul, on s'intéressera aussi au nombre de processeurs et à leur topologie, cf. (227).

Complexité au pire cas versus complexité en moyenne. La complexité au pire cas fournit une borne supérieure sur une ressource donnée. Mais, en pratique, la complexité en moyenne – par rapport à une distribution des entrées – peut être beaucoup plus significative (147). En revanche, l'analyse en moyenne est, en général de beaucoup plus difficile.

Complexité paramétrique. On peut aussi fixer un paramètre du problème et étudier sa complexité quand on fait varier les autres. C'est le champ d'investigation de la complexité paramétrique (52).

Axiomatique et théorie de la complexité. Mentionnons que Manuel Blum (24; 199) a défini un système axiomatique pour développer une théorie de la complexité qui soit indépendante des modèles de calcul.

1.8.4 Classes de complexité

Le titre de cette sous-section aurait pu être "*où l'ignorance crée l'abondance...*". A défaut de résultats significatifs, les problèmes sont classés comme le font les entomologistes (Aaronson et al.).

PTIME. La classe *PTIME* est la classe des problèmes qui sont calculables en temps polynômial. En d'autres termes, un problème est dans *PTIME* s'il existe un algorithme en temps polynômial qui le résout. La classe *PTIME* est robuste car elle est définissable par rapport à de nombreux modèles de calcul séquentiels. Elle comporte de nombreux problèmes complets :
- déterminer la valeur d'un circuit booléen monotone,
- satisfaisabilité des clauses de Horn,
- savoir si un langage hors-contexte est vide,

Enfin, *PTIME* est considérée comme la classe des problèmes "faisables" (en anglais "tractable"). Si cette affirmation est assez bien étayée, on peut cependant s'interroger sur le caractère raisonnable d'un algorithme en temps polynomial n^{100} par rapport à un algorithme en temps exponentiel en $2^{0.0000001 \times n}$.

Réduction entre problèmes. Problèmes complets.

NP. Face à la classe *PTIME*, il y a la très célèbre classe *NP* qui est définie à partir d'un modèle de calcul non déterministe. A chaque étape du calcul, la machine peut faire un choix et la solution du problème est trouvée s'il existe une séquence de choix qui conduit à la bonne solution. Entrons nous dans un pays du calcul imaginaire ? Pas tout à fait, car de nombreux algorithmes sont intrinsèquement non déterministes et d'autre part cette notion est mathématiquement viable et féconde.
La classe *NP* est la classe des problèmes qui sont calculables par une machine de Turing non déterministe en temps polynomial.
La notion et l'existence de problème NP complet est due à Cook (49) et Levin (146). Les travaux de Richard Karp puis le livre de Garey et Johnson (91) illustrent le foisonnement des problèmes NP-complets dans différents domaines comme en logique (3-SAT), en théorie des graphes (Clique, Vertex cover), en optimisation (Sac à dos),

La question *PTIME* $\stackrel{?}{=}$ *NP*. La célèbre question $PTIME = NP$ n'est pas que théorique car elle a des conséquences dans la vie quotidienne (51). Si $PTIME \neq NP$ alors les hypothèses de cryptographie seraient confirmées. Dans le cas contraire où $PTIME = NP$, et si nous en avons une démonstration raisonnable, les preuves dans certains systèmes logiques seraient alors possibles avec des applications remarquables, par exemple, en démonstration automatique.
Pourquoi cette question est-elle si ardue ? La question analogue avec le temps linéaire a été résolue : on sait que $LinearTime \neq NLinearTime$ ((177), 1986, cf. aussi le livre (9)) mais la

seule différence qu'on ait su prouver entre ces deux classes est qu'il y a un problème dans *NLinearTime* qui exige un temps déterministe en $O(n \log^* n)$. Comme on l'a vu au §1.8.3 (note de bas de page de l'exemple 1), cette fonction $\log^*$ est à croissance ultra lente. Un autre résultat surprenant est celui de Levin qui construit un algorithme optimal pour les problèmes *NP*qui est expliqué par Gurevich dans (111).

PSPACE. On peut aussi classifier les problèmes suivant la quantité de mémoire nécessaire à leur résolution. La classe *PSPACE* est la classe des problèmes calculables en espace polynômial. Le problème QBF (savoir si une formule quantifiée sur des variables booléennes est vraie) et de nombreux jeux, comme la généralisation du GO, ou encore le parsing des grammaires contextuelles sont des problèmes complets pour *PSPACE*. Ce qui est tout à fait remarquable ici, c'est que l'on sait que nombre de classes liées à l'espace sont égales :

$$PSPACE = NPSPACE = Parallel\ PTIME = APTIME$$

L'espace polynomial non déterministe *NPSPACE*ne donne rien de plus que l'espace polynômial déterministe *PSPACE* (194). De même, le temps polynômial *Parallel PTIME*sur machines RAM paralléles et celui *APTIME* sur machines alternantes (une extension forte du non déterminisme) (41) coïncident avec *PSPACE*.

La ribambelle. Nous avons juste entre ouvert la boîte de Pandore : il y a bien d'autres classes. A l'intérieur de *PTIME* mentionnons les classes NC^1 et *NLOGSPACE*. La première NC^1 est une clase uniforme de circuits booléens de taille polynomiale et de profondeur logarithmique (130). Un problème complet de NC^1 est l'évaluation d'une formule booléenne instanciée (39). La seconde classe *NLOGSPACE* contient des problèmes complets comme 2SAT ou encore le problème d'accessibilité dans un graphe orienté.
Cette courte exploration des classes de complexité peut être visualisée suivant leur relation d'inclusion ce qui donne la hiérarchie suivante :

$$AC^0 \subset NC^1 \subseteq LOGSPACE \subseteq NLOGSPACE \subset PTIME \subseteq NP \subseteq PSPACE \subseteq EXPTIME$$

La première inclusion, $AC^0 \subset NC^1$, est stricte. Furst, Saxe, and Spiser (88) et Ajtai (4) ont montré un résultat fondamental : La fonction parité n'est pas dans AC^0 mais elle est calculable dans *AC* qui est une classe incluse dans NC^1.

Pour en finir, Il y a des problèmes naturels qui sont plus compliqués que *PSPACE*, nous en avons cité un avec l'algèbre des nombres réels au §1.7.1

Résultats d'échange. La profusion des mesures de complexité soulève des questions sur le bien fondé des définitions données. Rassurons-nous, la complexité en temps semble pertinente et solide d'un point de vue pratique. D'autre part, il y a des résultats d'échange entre les ressources, et de hiérarchie à garder en tête : quelle que soit la fonction de complexité T,

$$TIME(T(n)) \subseteq NTIME(T(n)) \subseteq SPACE(T(n)) \subseteq NSPACE(T(n)) \tag{1.5}$$

où T est une fonction de complexité.

Une classe méconnue : *TIME*$(\mathbf{n}(\log \mathbf{n})^{\mathbf{O(1)}})$. Notons que cette classe est aussi close par composition comme l'est *PTIME*. Si cette classe est peu présente avec les problèmes combinatoires, en revanche, elle est capitale en analyse récursive. Richard Brent a, en effet, montré, 1976 (32), que toutes les fonctions de l'analyse (exponentielle, sinus, cosinus, fonction Γ,...) sont calculables en temps $n(\log n)^{O(1)}$ (il faut pour cela utiliser un algorithme de multiplication des entiers dans $n(\log n)^{O(1)}$, cf. Exemple 1 du §1.8.3).

1.8.5 Caractérisation des classes de complexité

Les classes de complexité sont définies à partir d ?un modèle de calcul et d ?une borne explicite sur la ressource considérée. La construction d"une représentation d ?une classe de complexité sans faire référence explicitement à un modèle de calcul borné a ouvert un champs de recherche appelé “Implicit Computational Complexity". Deux exemples historiques illustrent cette ligne de recherche. Le premier est une caractérisation de la classe *NP* en terme de relations définissables par des formules du second ordre par Fagin (2)et Jones-Selman (128)sur des structures finies. Le second est une caractérisation des fonctions primitives récursives comme l'ensemble des fonctions prouvablement totales dans l'arithmétique du premier ordre avec l'axiome d'induction restreint aux formules existentielles par Parsons. Parmi les travaux fondateurs qui ont proposé une caractérisation implicite de PTIME, il y a ceux sur la récursion bornée de Cobham (45), la logique avec point fixe de Immerman (122). A partir des années 90, d'autres modélisations sur des domaines infinis ont été construites : en logique du second ordre avec restriction de l'axiome de compréhension de Leivant (143), avec une récursion ramifiée de Bellantoni et Cook (12) et Leivant (144), avec une restriction au Lambda-Calcul de Leivant et Marion (145), avec des types linéaires de Girard (95), et enfin en logique du premier ordre avec une restriction sur le principe d'induction de Marion (156).

1.8.6 Diminuer la complexité en se contentant d'une bonne probabilité de réponse exacte

1.8.7 Complexité et représentation des objets

L'influence de la représentation sur la complexité des problèmes peut être considérable : on tremble à l'idée de faire une multiplication d'entiers représentés en chiffres romains... Nous sommes habitués aux représentations binaires et décimales des entiers. Mais il y en a bien d'autres. Considérons l'exemple de l'addition des entiers. On sait que les retenues peuvent se propager arbitrairement loin dans une addition. Ce fait simple empêche de faire des additions en parallèle sur les digits de même rang. Ce qui est bien dommage du point de vue complexité. Pourtant, cela est néanmoins possible. Un chercheur lithuano-américain, Algirdas Avizienis, a imaginé d'utiliser plus de chiffres que nécessaire (8). Ainsi, en base 2, peut-on utiliser les trois chiffres $-1, 0, 1$. Bien sûr, un nombre aura alors plusieurs représentations : par exemple, écrivant $\overline{1}$ au lieu de -1, le nombre 17 peut s'écrire 10001 ou $1001\overline{1}$ ou $101\overline{11}$ ou $11\overline{111}$ ou $1\overline{11111}$ ...Ce fait peut-être utilisé avec profit pour empêcher la propagation non bornée des retenues et permettre des additions en parallèle, cf. le livre de Jean-Michel Muller (167). Mentionnons que cette technique n'est pas pure théorie, elle est, en effet, implémentée dans les circuits des ordinateurs.

1.9 Calcul quantique

Observons que pour les systèmes quantiques, il est généralement admis qu'au niveau de la calculabilité, on ne peut calculer plus qu'avec une machine de Turing (68). Par contre, au niveau de la complexité, les algorithmes quantiques permettent prouvablement de résoudre certains problèmes plus rapidement (107).

1.9.1 Inserer le texte de Simon Perdrix

1.9.2 ...

1.9.3 Inserer le texte de Simon Perdrix

1.10 Une opérationnalité efficace : les automates finis

Les automates finis peuvent être considérés comme issus de considérations algébriques ou issus de considérations algorithmiques avec une origine multiple : les automates de calcul, les machines de Turing, les neurones formels, les chaînes de Markov. Nous privilégierons plutôt la vision algorithmique dans cette section. Il est usuel de proposer comme date de début des travaux sur les automates finis la date de publication du théorème de Kleene, soit 1954. Depuis cette date, un vaste courant de recherche s'est intéressé aux automates finis et à leurs applications avec une activité très forte dans les années 1960-1990 et de nombreux et fructueux résultats. Désormais, les automates finis font partie de la culture de la communauté informatique. Cependant, les recherches sur les automates finis et leurs applications restent très actives avec des résultats importants très récents avec des applications en bio-informatique, en traitement du langage naturel, en systèmes d'information parmi d'autres. Nous nous proposons de parcourir les résultats fondamentaux et les principales applications. Nous nous excusons par avance pour les oublis dus au choix des auteurs. Les références principales dont nous nous sommes servis sont : les livres de Hopcroft et Ullman dont (120) ; les chapitres 1 à 5 du volume B de (228) par Perrin, Berstel et Boasson, Salomaa, Thomas et Courcelle ; de nombreux chapitres de (189) ; le livre de Sakarovitch (192) ; un ouvrage récent sur les automates pondérés (75) ; un ouvrage collectif sur les automates d'arbres (48).

1.10.1 Les automates finis de mots

Les automates finis de mots sont un modèle de calcul très simple : des états en nombre fini, des règles en nombre fini décrivant le changement d'état à effectuer lorsqu'on lit une lettre et des conditions initiales et finales. Le modèle de base considère des mots qui sont des séquences finies de lettres d'un alphabet fini. Formellement, étant donné un alphabet fini de lettres A, un *automate fini* $\mathcal{M}$ est défini par un ensemble fini d'états Q, un sous-ensemble I de Q des états initiaux, un sous-ensemble F de Q des états finaux, un ensemble de transitions Δ inclus dans $Q \times A \times Q$. L'automate $\mathcal{M}$ est décrit par un quadruplet (Q, I, F, Δ). On peut également voir un automate $\mathcal{M}$ comme un graphe avec les noeuds dans Q, les arcs dans Δ et deux ensembles particuliers de noeuds que sont les noeuds initiaux et finaux. Il est d'usage de faire une représentation graphique d'un automate en choisissant une convention pour représenter les noeuds initiaux et finaux. Un automate est *complet* (ou complètement spécifié) si pour tout couple (q, a) dans $Q \times A$, il existe une règle (q, a, q') dans Δ. Un automate est *déterministe* si pour tout couple (q, a) dans $Q \times A$, il existe au plus une règle (q, a, q') dans Δ.

Les automates finis ainsi définis sont considérés comme des "accepteurs" (ou "reconnaisseurs") car ils répondent oui ou non à la question : ce mot est-il accepté (reconnu) par l'automate ? Pour cela, il reste à définir comment cette réponse peut-être calculée. Un *calcul* de $\mathcal{M}$ sur un mot $u = a_1 \dots a_n$ est un mot $q_0 q_1 \dots q_n$ tel que : q_0 est un état initial dans I, et, pour tout $i < n$, (q_i, a_i, q_{i+1}) est une règle dans Δ. Notons que dans un automate complet (respec-

tivement déterministe), il existe toujours au moins un (respectivement au plus un) calcul pour tout mot u. Un mot u est accepté par $\mathcal{M}$ si il existe un calcul de $\mathcal{M}$ sur u dont le dernier état est final, un tel calcul est appelé calcul réussi. Le langage des mots acceptés par $\mathcal{M}$ est noté $L(\mathcal{M})$. Un langage L (un ensemble de mots) est *reconnaissable* si il existe un automate fini $\mathcal{M}$ tel que $L = L(\mathcal{M})$.

Nous présentons sans démonstration une sélection de résultats de base sur les automates finis de mots. Pour tout automate, il existe un automate complet qui reconnaît le même langage. Pour tout automate à n états, il existe un automate déterministe complet à au plus 2^n états qui reconnaît le même langage. On peut noter que les calculs dans un automate sont effectués par une lecture de gauche à droite du mot et une lettre est consommée à chaque application de règle. Une première extension consiste à autoriser un changement d'état sans consommer de lettre. On parle d'automate avec ϵ-transitions (ou ϵ-règles). Il existe un algorithme de construction d'un automate reconnaissant le même langage qui ne contient pas d'ϵ-transitions et donc l'expressivité de ces automates correspond à la classe des langages reconnaissables. Une autre extension consiste à permettre d'alterner le sens de lecture du mot pendant le calcul. Ici encore l'expressivité reste inchangée.

Une propriété importante des langages reconnaissables s'exprime à l'aide de *lemmes de pompage*. Dans sa version première (des lemmes plus forts ont été proposés), ce lemme affirme que, pour tout langage reconnaissable, il existe une constante k, telle que tout mot du langage de longueur supérieure à k possède une factorisation avec un facteur qui peut être itéré ou enlevé en restant dans le langage. Ce lemme permet de montrer facilement qu'un langage tel que $\{a^n b^n \mid n \geq 0\}$ n'est pas régulier.

La classe des langages reconnaissables est *close par les opérations booléennes*, à savoir intersection, union et complémentation : par exemple, l'intersection de deux langages reconnaissable est reconnaissable. Elle est close par homomorphisme et homomorphisme inverse. Le vide d'un langage est décidable.

Le résiduel à gauche $u^{-1}L$ d'un langage L relativement à un mot u est le langage constitué des mots v qui sont tels que u suivi de v, i.e. uv, est un mot de L. On peut définir une relation $\equiv_L$ par $u \equiv_L v$ si $u^{-1}L = v^{-1}L$, i.e. u et v ont les mêmes prolongements dans L. Cette relation est une relation d'équivalence qui est stable par concaténation, c'est donc une relation de congruence. Le *théorème de Myhill-Nerode* affirme que L est un langage régulier si et seulement si le nombre de classes d'équivalence de $\equiv_L$ est fini. Ce théorème implique principalement l'existence d'un automate déterministe minimal pour tout langage régulier L au sens du nombre minimal d'états. Ce nombre minimal d'états est le nombre de classes de congruence de $\equiv_L$. Des algorithmes de construction de l'automate minimal ont été définis : un algorithme dérivé de la preuve du théorème, l'algorithme de Hopcroft par raffinement de partition en partant d'un automate déterministe et de la partition $(F, Q - F)$, un algorithme par passage au miroir et déterminisation du miroir.

Nous présentons maintenant différents formalismes qui sont prouvés équivalents aux automates finis du point de vue expressivité. Ceci démontre l'importance des automates finis et de la classe des langages reconnaissables. Nous présentons ensuite certains résultats de complexité qui eux dépendent du formalisme choisi.

Grammaires régulières. Les automates privilégient le point de vue "accepteur" ou "reconnaisseur" et les considérations algorithmiques. Les grammaires ont été introduites pour une étude des classes de langages principalement dans le domaine des langues naturelles.

Elles privilégient le point de vue "générateur" en décrivant comment générer les mots d'un langage par des règles de production (ou de réécriture). Formellement, étant donné un alphabet fini de lettres A, une *grammaire* $\mathcal{G}$ est définie par un ensemble fini de non terminaux (ou variables) N, un ensemble P de règles de production de la forme $l \to r$ où l et r sont des mots sur l'alphabet $A \cup N$ et un non terminal particulier souvent noté S (pour "sentence") appelé axiome. La grammaire $\mathcal{G}$ est décrite par un triplet (N, P, S). On génère un mot en partant de S et en appliquant des règles de P jusqu'à obtenir un mot u ne contenant que des lettres de A. Le langage des mots générés par $\mathcal{G}$ est noté $L(\mathcal{G})$. Sans restriction sur la forme des règles, on a les grammaires de type 0 de Chomsky qui génèrent les langages récursivement énumérables, i.e. reconnaissables par machine de Turing (voir Section 1.2.4). Une restriction très forte est de considérer les grammaires de type 3 également appelées grammaires régulières : une *grammaire est régulière* si toutes les règles de P ont leur membre gauche qui se réduit à un non terminal dans N et leur membre droit est, soit une lettre de A, soit une lettre de A suivie d'un non terminal dans N. Il est facile de montrer de façon constructive que la classe de langages générés par des grammaires régulières est la classe des langages reconnaissables.

Expressions régulières. Après les points de vue accepteur et reconnaisseur, nous prenons maintenant le point de vue "descriptif" avec les expressions régulières introduites par Kleene (131). Celles-ci permettent de décrire par une expression à la syntaxe précise un ensemble de mots, c'est-à-dire un langage. Cette syntaxe est souvent spécifique à chaque langage informatique. Nous prenons ici le point de vue formel et considérons les opérations sur les langages que sont la concaténation notée . (souvent oublié) et définie par $X.Y = XY = \{w \mid w = uv, u \in X, v \in Y\}$; l'union notée $+$ en notation infixée ; l'étoile $*$ en notation postfixée et définie par $X^* = \{w \mid w = u_1 \dots u_n, n \geq 0, u_1, \dots, u_n \in X\}$, soit encore l'union infinie des puissances d'ordre $n \geq 0$ d'un langage. On peut ainsi décrire comment sont "construits" les mots du langage en utilisant "suivi de" pour la concaténation, "ou" pour l'union et "un certain nombre de" pour l'étoile. Un langage est *rationnel* si il peut être décrit par une expression rationnelle.

Le théorème de Kleene, présent dans le papier originel, montre l'égalité entre la classe des langages reconnaissables et la classe des langages réguliers. Au vu du grand nombre d'applications, les transformations entre automates et expressions régulières ont été et demeurent beaucoup étudiées. Pour *transformer une expression régulière en un automate équivalent*, une première solution est d'utiliser une construction sur les automates pour chaque opérateur ., + et $*$. Ceci se fait facilement avec des automates finis avec ϵ-transitions. Cette construction n'est pas optimale car on peut avoir beaucoup d'ϵ-transitions et leur élimination peut être coûteuse. Une construction d'un automate fini sans ϵ-transitions, appelé automate des positions ou automate de Glushkov, équivalent de taille quadratique par rapport à la taille de l'expression régulière est attribuée à Glushkov (96). Cette construction aux très nombreuses applications a été reprise, améliorée et particularisée pour des classes d'expressions. Citons, entre autres, McNaughton et Yamada (163), Berry et Sethi (16), Brüggeman-Klein (33; 34). Pour *transformer un automate en une expression régulière équivalente*, une solution est d'écrire un système d'équations rationnelles de la forme $X_i = \sum_j X_j a_{ij}$ ou $X_i = \sum_j X_j a_{ij} + \epsilon$ où la variable X_i capture les mots pouvant arriver dans l'état q_i et a_{ij} une lettre de l'alphabet, puis de résoudre ce système en utilisant des substitutions et le fait que l'unique solution

de l'équation $X = XY + Z$ lorsque $\epsilon \notin Y$ est ZY^*. Mais, Ehrenfeucht et Zeiger (79) définissent des automates à n états pour lesquels la plus petite expression régulière équivalente contient un nombre exponentiel d'occurrences de lettres de A dans l'expression.

La question du nombre d'étoiles nécessaires pour définir un langage reconnaissable s'est posée très tôt. Une première réponse a été de montrer que la hauteur d'étoiles (le nombre d'imbrications dans l'expression) ne peut pas être bornée (Eggan (78), amélioré dans Dejean et Schützenberger (64)). Un deuxième résultat plus difficile, obtenu par Hashiguchi (116), fut de proposer un algorithme pour déterminer la hauteur d'étoile d'un langage régulier. Des améliorations algorithmiques ont été apportées depuis mais la complexité reste exponentielle et impraticable.

Logique monadique du second ordre (MSO). Un mot peut être représenté par une structure du premier ordre avec un ensemble de positions qui est un intervalle $[1, n]$ d'entiers, une notion de successeur, un ordre naturel et pour chaque lettre de l'alphabet l'ensemble des positions du mot correspondant à cette lettre. On peut alors considérer un langage du premier ordre où les variables, notées x, y, ..., représentent des entiers (des positions) et des formules atomiques de la forme $x + 1 = y$ (la position suivante de x est y), $x < y$ (la position x précède y), ou encore $P_a(x)$ (en position x on trouve la lettre a). On considère également des variables du second-ordre, notées X, Y, ..., qui représentent des ensembles (finis) de positions. Les quantifications peuvent porter sur des variables du premier ordre et du second ordre. Un langage est *définissable en MSO* si il correspond à l'ensemble des mots qui satisfont une formule ϕ sans variable libre. Büchi (37) et Elgot (82) ont montré que les langages définissables en MSO sont les langages reconnaissables. Cette logique est donc décidable car on peut décider de la satisfiabilité d'une formule en calculant un automate et en décidant du vide pour cet automate. Ce résultat permet donc de résoudre la question de la décidabilité de la théorie des entiers avec successeur et quantification sur les ensembles d'entiers (voir Section 1.7.1). Ce résultat peut être vu comme un cas particulier d'un résultat similaire, obtenu par Büchi (36) et McNaughton (161), pour la logique MSO où on considère des positions entières ce qui revient à interpréter sur des mots infinis. Pour la preuve d'équivalence entre langages déinissables en MSO et langages reconnaissables, le passage d'une formule vers un automate se fait par une méthode d'élimination de quantificateurs. Chaque alternance de quantificateur nécessite un calcul du langage complémentaire et donc une exponentiation. Ceci est intrinsèque car le problème est démontré non élémentaire. Le passage de l'automate à une formule consiste à décrire dans une formule MSO le fonctionnement de l'automate, ce qui ne pose pas de difficulté.

Automates alternants. Introduits dans (41) et (35), ils généralisent les automates finis non déterministes tout en gardant le même pouvoir d'expression et en permettant une construction simple pour la complémentation. Supposons que dans un automate non déterministe, on dispose d'un choix avec deux règles (q, a, q_1) et (q, a, q_2). Pour qu'un mot u décomposé sous la forme $u = av$ soit reconnu, il suffit qu'il existe un calcul réussi partant de q_1 lisant v ou qu'il existe un calcul réussi partant de q_2 lisant v. D'autre part, pour que u ne soit pas reconnu, il faut vérifier que tous les calculs ne sont pas réussis. L'idée de base des automates alternants est d'autoriser dans leur définition et dans la définition de calcul réussi ces deux types de comportement : disjonctif où un des calculs doit vérifier une condition et conjonctif où tous les calculs doivent vérifier une condition.

Il existe plusieurs définitions équivalentes des automates alternants utilisant des types d'état existentiel et universel ou utilisant des formules logiques (avec ou sans négation) pour les membres droits de règles. Nous ne détaillons pas ces formalismes mais notons que, quel que soit le formalisme choisi, la preuve de clôture par complémentation est immédiate. On montre également (41) que la classe des langages reconnus est la classe des langages reconnaissables et qu'il existe des automates alternants à k états pour lesquels un automate complet déterministe équivalent a au moins 2^{2^k} états.

Nous terminons par quelques résultats de complexité. La complexité dépend de la représentation choisie en sachant également que nous avons donné dans ce qui précède des éléments sur la complexité de passage entre formalismes. Par exemple, décider si un mot satisfait une formule MSO se fait en temps linéaire par rapport à la taille du mot et la taille d'un automate représentant la formule. Mais on sait que la construction de l'automate est non élémentaire. Nous donnons la complexité de quelques problèmes : la décision de l'appartenance d'un mot à un langage est DLOGSPACE-complet avec un automate déterministe en entrée, NLOGSPACE-complet avec un automate fini en entrée, P-complet avec un automate alternant en entrée ; la décision du vide d'un langage est NLOGSPACE-complet avec un automate fini (déterministe ou non) en entrée, P-complet avec un automate alternant en entrée ; la décision de l'équivalence de deux langages est NLOGSPACE-complet avec deux automates déterministes en entrée, P-complet avec deux automates non déterministes ou deux automates alternants en entrée.

1.10.2 Les extensions des automates de mots

Automates pondérés et séries rationnelles

Un automate détermine l'appartenance d'un mot à un langage reconnaissable avec une réponse booléenne. Une extension naturelle est d'associer à un mot une probabilité d'appartenance à un langage. Ceci peut être réalisé à l'aide d'automates probabilistes introduits dans (180). Le principe est d'associer des poids réels positifs aux états initiaux, aux états finaux et aux règles de l'automate. En introduisant des conditions de normalisation adéquates de ces poids ainsi que des conditions d'accessibilité et de co-accessibilité des états, un automate probabiliste définit une loi de probabilité sur l'ensemble des mots sur un alphabet A, les mots de probabilité non nulle définissant un langage reconnaissable. On peut noter que les automates déterministes probabilistes sont moins expressifs que les automates probabilistes. Par contre les automates probabilistes correspondent aux modèles de Markov cachés (76).

Pourquoi se limiter à des calculs dans les nombres réels dans l'intervalle $[0, 1]$ alors que les automates devraient être, par exemple, capables de compter. On peut ainsi définir les automates pondérés avec des poids dans un semi-anneau $(K, +, \times, 0, 1)$ qui vont permettre de définir des applications de l'ensemble des mots sur un alphabet A dans le semi-anneau K. Ces applications sont appelées séries reconnaissables. Elles ont été montrées équivalentes aux séries rationnelles définies de façon algébrique en théorie des langages formels (18; 192; 75).

Automates à piles et langages algébriques

Dans la hiérarchie de Chomsky et dans l'étude du langage naturel, une classe très importante est celle des grammaires de type 2, encore appelées grammaires à contexte libre ou

grammaires algébriques. Une grammaire $\mathcal{G}$ est algébrique si tous ses membres gauches sont réduits à un non terminal. Cette restriction implique que la génération d'un mot est effectuée par réécriture des non terminaux indépendamment du contexte. La classe des langages algébriques contient strictement la classe des langages reconnaissables : le langage $\{a^n b^n \mid n \geq 0\}$, le langage des expressions bien parenthésées sont algébriques et non reconnaissables. En effet, il est possible de contrôler la génération simultanée de parenthèse ouvrante et fermante et les imbrications. Ceci implique pour reconnaître de tels langages, d'ajouter aux automates la capacité de compter et de vérifier qu'à toute parenthèse ouvrante correspond bien une parenthèse fermante. Ceci se fait par l'ajout d'une pile à l'automate fini et la possibilité pour les règles d'empiler ou de dépiler des symboles sur la pile. Nous ne détaillerons pas le formalisme des automates à piles. Les langages algébriques et les automates à piles ont été très largement étudiés à cause de leurs très nombreuses applications : langages de programmation et langue naturelle, par exemple. Des points d'entrées pour la très vaste littérature peuvent être les articles sur les grammaires algébriques de (228; 189). Mais on peut noter que certains résultats importants sont postérieurs. Par exemple, l'équivalence d'automates à piles déterministes a été montrée décidable à la fin des années 90 par Sénizergues (201).

Automates avec sorties

Une autre extension des automates consiste à permettre aux automates de lire leur entrée tout en calculant un mot de sortie. Cette extension naturelle qui prend son origine dans les machines de Turing a donné lieu aux machines de Moore, de Mealy et de façon plus générale aux transducteurs. Un transducteur est un automate de mots auquel on ajoute à toute règle de la forme (q, a, q') un mot u de sortie (éventuellement vide) sur un alphabet B. Une règle peut être notée sous la forme $(q, a/u, q')$. Le fonctionnement est alors modifié en ajoutant le mot u à la sortie courante lorsque l'on active cette règle. Un transducteur définit ainsi une relation reconnaissable. Ce domaine a été largement étudié en théorie des langages formels et nous renvoyons à l'ouvrage de Sakarovitch (192) qui contient résultats essentiels et pointeurs sur la littérature.

Différentes classes de transducteurs peuvent être considérées selon que l'on autorise ou pas des ϵ-règles (possibilité de générer un mot de sortie sans lire de lettre de l'entrée), selon le cas déterministe ou non. L'équivalence des transducteurs non déterministe sans ϵ-règle est indécidable (104). L'équivalence de transducteurs déterministe est décidable (109). Ce résultat reste vrai si on restreint le domaine à un langage algébrique avec un calcul en temps polynomial, ceci peut être établi comme conséquence des résultats de la thèse de Plandowski (182). Enfin, le résultat de Sénizergues a été étendu aux transducteurs à piles déterministes (200).

1.10.3 Automates finis pour les structures discrètes

Automates d'arbres

La structure d'arbre est omniprésente en informatique : analyse de programmes, arbres et-ou, arbres de classification, arbres et documents structurés, arbres phylogénétiques, ... Les définitions d'arbres sont diverses selon le cadre choisi. Nous considérons ici les arbres qui ont une racine, qui sont ordonnés (les fils d'un noeud sont ordonnés), qui sont étiquetés (à chaque noeud est associé un symbole d'un alphabet fini) et qui sont d'arité fixée (le nombre de fils

d'un noeud est déterminé par le symbole en ce noeud). Nous supposons donc l'existence d'un alphabet fini Σ où chaque symbole a une arité fixée entière qui détermine le nombre de fils. L'ensemble des arbres sur Σ peut être défini récursivement par : tout symbole d'arité 0 est un arbre, si $t_1, \ldots, t_n$ sont des arbres et si f est un symbole d'arité n alors $f(t_1, \ldots, t_n)$ est un arbre. On peut voir un arbre comme un graphe avec une racine, un arc entrant par noeud, un nombre d'arcs sortants respectant l'arité du symbole du noeud et un ordre sur les arcs sortants. On représente généralement un arbre avec la racine en haut et les feuilles (les noeuds sans fils qui correspondent à des symboles d'arité 0) en bas.

Étant donné un alphabet fini Σ, un *automate (ascendant) d'arbres* $\mathcal{M}$ est défini par un ensemble fini d'états Q, un sous-ensemble F de Q des états finaux, un ensemble de transitions Δ de règles de la forme $a \to q$ avec a lettre de Σ d'arité 0 et q état de Q ou de la forme $f(q_1, \ldots, q_n) \to q$ avec avec f lettre de Σ d'arité n et $q_1, \ldots, q_n, q$ états de Q. L'automate $\mathcal{M}$ est décrit par un triplet (Q, F, Δ). On peut noter qu'il n'y a pas d'états initiaux car les règles de la forme $a \to q$ servent à initier un calcul. En effet, un *calcul* de $\mathcal{M}$ sur un arbre t est un arbre r qui a le même ensemble de noeuds que t, qui est étiqueté par des symboles dans Q et cet étiquetage respecte les règles de $\mathcal{M}$ sur t. Un automate est *déterministe* si pour tout symbole f d'arité n et tout n-uplet d'états $(q_1, \ldots, q_n)$, il existe au plus une règle $f(q_1, \ldots, q_n) \to q$ dans Δ. Notons que dans un automate déterministe, il existe toujours au plus un calcul pour tout arbre t. Un arbre t est accepté par $\mathcal{M}$ si il existe un calcul r de $\mathcal{M}$ sur t dont l'état à la racine de r est final, un tel calcul est appelé calcul réussi. Le langage des mots acceptés par $\mathcal{M}$ est noté $L(\mathcal{M})$. Un langage L (un ensemble d'arbres) est *reconnaissable* si il existe un automate fini $\mathcal{M}$ tel que $L = L(\mathcal{M})$.

La plupart des propriétés s'étendent du cas des mots au cas des arbres : déterminisation, clôture par les opérations booléennes, lemme de pompage, décision du vide. La notion de grammaire régulière d'arbres peut être définie. Des expressions rationnelles peuvent être définies en introduisant une opération de concaténation pour les arbres qui consiste à ajouter un arbre dans un contexte (un arbre avec une feuille particulière autorisant l'ajout) et un théorème de Kleene peut être montré. Un théorème de Myhill-Nerode et un algorithme de minimalisation peuvent être définis. La logique MSO peut être étendue pour être interprétée sur des arbres finis, elle définit la classe des langages reconnaissables d'arbres et est donc décidable (223). Ce résultat a été étendu aux arbres infinis (et les automates correspondants) pour obtenir un résultat fondamental aux nombreuses applications : le *théorème de Rabin (Rabin's Tree Theorem)* (185) qui établit que la logique monadique du second-ordre de l'arbre infini binaire est décidable. Les différences principales entre automates d'arbres et de mots sont :

Automates descendants déterministes : on peut considérer des automates, appelés *automates descendants*, qui commencent le calcul à la racine et le terminent aux feuilles. Il suffit d'inverser le sens des règles et d'échanger les états finaux en états initiaux pour définir un automate descendant à partir d'un automate d'arbres. Si les automates descendants et les automates ascendants (déterministes ou non) définissent les langages reconnaissables d'arbres, les automates descendants déterministes (pour un état q et une lettre f il existe au plus une règle $q \to f(q_1, \ldots, q_n)$) reconnaissent une sous-classe stricte des langages reconnaissables. Le lecteur se persuadera facilement que le langage reconnaissable contenant les deux arbres $f(a, b)$ et $f(b, a)$ ne peut être reconnu par un automate descendant déterministe.

Les "tree walking automata" : dans ce modèle, on peut tester si la position courant cor-

respond à une feuille, la racine, un fils droit ou un fils gauche d'un noeud. Les règles permettent d'annoter la position courante par un état et de rester dans la même position, de monter au noeud père ou de descendre au fils droit ou fils gauche de la position courante. Un calcul réussi considère une configuration initiale avec un état initial à la racine de l'arbre, des configurations obtenues en appliquant les règles et une configuration finale avec un état final à la racine de l'arbre. Si il est facile de voir que les langages reconnus sont reconnaissables il n'a été démontré que récemment que l'inclusion est stricte (il existe des langages reconnaissables non reconnus dans ce modèle) et que le déterminisme restreint le pouvoir d'expression. Ces résultats ont été obtenus par Bojanczyk et Colcombet (25; 26).

La linéarité : la clôture par morphisme des langages reconnaissables n'est pas vérifiée dans les arbres. Un morphisme est *linéaire* si on interdit la duplication de variables. La classe des langages reconnaissables est close par morphisme linéaire. Mais pas dans le cas général. En effet, si on considère un langage reconnaissable $\{g(f^n(a)) \mid n \geq 0\}$ et un morphisme qui transforme $g(x)$ en $h(x, x)$, l'image du langage est $\{h(f^n(a), f^n(a)) \mid n \geq 0\}$ qui n'est pas un langage reconnaissable d'arbres.

La logique du premier ordre : la logique MSO correspond aux langages reconnaissables dans le cas des mots et le cas des arbres. Si on se restreint au premier ordre, la situation est différente. En effet, dans le cas des mots, la classe des langages définissables dans FO correspond à la classe des langages sans étoile (162), i.e. les langages qui peuvent être définies par des expressions rationnelles avec les opérateurs de concaténation, d'union et de complémentation. Les langages sans étoile correspondent aux langages apériodiques (198). La situation est bien plus complexe pour la logique FO dans le cas des arbres et une caractérisation décidable de langages d'arbres définissables en FO n'a été obtenue que récemment par Benedikdt et Ségoufin (13).

Le cas des graphes

Après avoir étudié les mots, puis les arbres avec les automates adéquats, l'étape suivante naturelle est de considérer les graphes. Cependant, les graphes sont intrinsèquement plus compliqués et *il n'est pas possible de définir une bonne notion d'automate*. Une bonne notion signifiant obtenir des algorithmes et propriétés de décision, une correspondance entre les points de vue automates, grammaires et logiques. Un point d'entrée sur ces questions peut être le chapitre 5 du volume B du Handbook (228) par Courcelle qui a, par ailleurs, écrit de nombreux articles fondamentaux sur les grammaires et logiques pour les (hyper-)graphes et qui prépare un ouvrage (58) sur ces sujets.

1.10.4 Automates et applications

Nous proposons pour conclure un choix (de domaines) d'applications des automates finis.

Algorithmes de filtrage : ils considèrent la question de rechercher des facteurs dans des mots. L'algorithme de Knuth, Morris and Pratt (133) a été le premier algorithme linéaire pour le filtrage. Cet algorithme a été conçu avec une vision "automates" du problème de filtrage. Au vu de l'importance applicative de la question (pensez aux `grep` et `egrep` de UNIX), de nombreux algorithmes, parfois à base d'automates, ont été définis pour le

filtrage de plusieurs facteurs, le filtrage d'expressions régulières. Nous renvoyons le lecteur au Chapitre "Automata for matching patterns" de Crochemore et Hancart de (189) pour une présentation de ces algorithmes. Les recherches sur ces questions sont toujours actives en bio-informatique et sur le filtrage approximatif.

Codes, cryptographie et théorie de l'information : les codes sont des langages formels (voir Berstel et Perrin (17)) aux propriétés particulières pour des applications en cryptographie et en théorie de l'information. Dans ce cadre, les automates ont été utilisés pour étudier la décidabilité et la complexité de problèmes de décision comme, par exemple, décider de l'appartenance d'un langage à une classe de codes.

Analyse de langages : elle considère le problème de tester l'appartenance d'un mot à un langage et, dans le cas d'une réponse positive à produire une preuve d'appartenance. Dans le cadre des langages algébriques, il s'agit de construire un arbre d'analyse (qui représente une dérivation dans la grammaire) pour un mot du langage. Des algorithmes, souvent basés sur des (classes de) automates à pile ont été définis pour des classes de langages algébriques ou des formes particulières de grammaires. Ce problème très étudié est fondamental pour la compilation, l'évaluation dans les langages de programmation.

Langage naturel : les travaux sur les langages algébriques et les algorithmes d'analyse ont été considérés également pour les grammaires en langage naturel. D'autres travaux concernent l'étude de l'expressivité des différents formalismes introduits pour modéliser le langage. Par exemple, le théorème de Gaifman (10) montre l'équivalence entre grammaires catégorielles et grammaires algébriques. La question posée par Chomsky de l'équivalence entre grammaires de Lambek et grammaires algébriques a été montrée par Pentus (181). L'introduction de formalismes pour une meilleure modélisation du langage naturel est constante. On peut, par exemple, citer les grammaires d'arbres adjoints et les grammaires de dépendance. Enfin, les transducteurs de mots et d'arbres ont été très utilisés dans le domaine, en particulier, pour la traduction automatique. Actuellement, de nombreux travaux considèrent des méthodes statistiques à base de corpus. Dans ce cadre, les grammaires et automates probabilistes sont utilisés (voir ci-après).

Extraction d'information et systèmes questions/réponses : l'extraction d'information consiste à extraire des informations structurées, par exemple une table des nom, prénom, institution, date d'arrivée à partir de documents textuels ou HTML décrivant des groupes de recherche. Les systèmes questions/réponses ambitionnent de trouver des réponses à des questions, souvent factuelles comme trouver la date d'un événement, en interrogeant le Web. Des solutions à base d'automates et de transducteurs ont été étudiées et développées dans le cadre de ces applications. Le lecteur pourra consulter comme exemple Poibeau (183).

Bases de données XML : XML est un standard pour l'échange de données et documents. Les documents XML ont une structure d'arbre au sens défini précédemment mais avec la spécificité nouvelle suivante : un noeud a un nombre potentiellement non borné de fils. Des classes d'automates et de transducteurs ont été définies et étudiées pour ces arbres ou leurs représentations linéaires avec balises ouvrantes et fermantes. La forme des documents peut être contrainte par des schémas. Ces schémas peuvent être définis par des expressions régulières particulières réactivant les travaux sur les algorithmes de tranformations entre expressions et automates. Ces schémas peuvent être définis par des

langages reconnaissables d'arbres, c'est le cas de RelaxNG. Le lecteur intéressé peut consulter le chapitre 8 de (48) comme point d'entrée sur ce domaine.

Théorie des modèles finis : elle étudie les logiques pour les structures finies et tire son origine des bases de données, de la théorie des langages formels et de la complexité. Une référence peut être le livre d'introduction par Libkin (150). Les automates finis sont, comme nous l'avons signalé, des algorithmes de décision pour la logique MSO sur les mots ou sur les arbres. Des applications actuelles concernent l'étude de l'expressivité et de la complexité des langages de requêtes dans les bases de données XML en lien avec l'étude de l'expressivité et de la complexité de fragments de la logique MSO des arbres. On peut noter également, un courant de recherche sur les structures automatiques qui concerne l'étude des propriétés d'objets ou de relations entre objets qui peuvent être codées dans (des fragments de) la logique MSO (190).

Réécriture et déduction automatique : La réécriture est un processus naturel de calcul comme nous l'avons vu pour les grammaires. Pour les systèmes de réécriture de mots et d'arbres, les questions de terminaison et de confluence des calculs se posent. Une introduction aux systèmes de réécriture d'arbres par Dershowitz et Jouannaud est présentée dans le chapitre 6 de (228). Les automates d'arbres peuvent être utilisés pour résoudre des questions de décidabilité pour ces systèmes et pour fournir des algorithmes associés si possible. Par exemple, Dauchet et Tison ont démontré que la théorie de la réécriture avec des règles sans variables dans les termes est décidable. Ce résultat est présenté dans le chapitre 3 de (48). Ce chapitre contient également des applications des automates d'arbres au typage, à la réécriture, au filtrage et à l'unification.

Automates probabilistes et modèles de Markov cachés : les automates probabilistes (non déterministes) correspondent aux modèles de Markov cachés dont les applications sont très nombreuses : traitement du signal, traitement des séquences biologiques, extraction d'information parmi d'autres. Ces applications sont rendues possibles par l'existence d'algorithmes efficaces pour l'inférence (le calcul de la probabilité d'une séquence), l'annotation (le calcul du chemin le plus probable dans l'automate) par l'algorithme de Viterbi, l'apprentissage des poids à partir de corpus avec une structure d'automate fixée par l'algorithme de Baum-Welch. Le lecteur peut consulter l'ouvrage de Manning et Schütze (153) orienté langage naturel et l'ouvrage de Durbin et al (77) orienté bio-informatique pour les modèles de Markov cachés comme pour les grammaires algébriques probabilistes.

Grammaires algébriques probabilistes : une grammaire algébrique probabiliste se définit en ajoutant des poids réels positifs sur les règles de transition avec des conditions de normalisation. Une telle grammaire, contrairement au cas régulier, ne définit pas nécessairement une loi de probabilité. Il n'a été montré que récemment qu'il était décidable si une grammaire algébrique probabiliste définit une loi de probabilité par Etessami et Yannakakis (83). L'analyse probabiliste calcule l'arbre d'analyse le plus probable ou les k meilleurs arbres d'analyse pour un mot et une grammaire algébrique probabiliste. Il est important de noter que l'algorithme inside-outside étend l'algorithme de Viterbi des séquences aux arbres d'analyse pour le calcul de l'arbre d'analyse le plus probable. De même, on peut étendre l'algorithme de Baum-Welch pour apprendre, à structure fixée, les poids d'une grammaire algébrique probabiliste à partir de corpus constitués de

couples séquence, arbre d'analyse. Les applications principales concernent principalement langage naturel et bio-informatique.

Inférence grammaticale : pour les modèles de Markov cachés et les grammaires algébriques probabilistes, l'algorithme de Baum Welch permet d'apprendre les poids associés aux transitions ou aux règles, une structure d'automates ayant été choisie a priori. L'inférence grammaticale se pose la question d'inférer la structure d'automate (ou de grammaire) la plus adaptée à un corpus. La question est difficile car les langages reconnaissables ne sont pas apprenables à partir de mots du langage (une trop grande généralisation ne pouvant être évitée). Par contre, des algorithmes ont pu être proposés dans le cas où on dispose de mots du langage et de mots n'appartenant pas au langage. La plupart des algorithmes construisent l'automate déterministe minimal et sont basés sur le théorème de Myhill Nérode. Pour le cas probabiliste, on apprend à partir de mots tirés à partir de la distribution à apprendre. Les algorithmes précédents peuvent être étendus pour apprendre les distributions définies par des automates déterministes probabilistes. Récemment de nouveaux algorithmes basés sur une vision algébrique ont été conçus par Denis et al (66), ils permettent d'étendre la classe des lois de probabilité apprenables.

1.11 Complexité algorithmique de l'information

Utilisant le code ASCII dans lequel chaque lettre est codée par 8 bits, un texte de taille peut être représenté par $8n$ bits. Mais, certaines lettres peuvent être absentes ou, au contraire, être plus fréquentes que les autres. En codant les lettres très fréquentes par peu de bits quitte à devoir coder les autres par bien davantage, on peut obtenir des codages du texte par beaucoup moins que $8n$ bits. Jusqu'où peut-on aller de la sorte ? C'est la question qu'a résolue Claude Shannon en 1948 dans un très célèbre article (203) qui fonde un nouveau sujet : la théorie quantitative de l'information, sujet aux applications innombrables dans tous les moyens de communication (téléphone, fax, etc.). Shannon introduit une notion d'entropie : si les fréquences des différentes lettres dans un texte sont $f_1, \ldots, f_n$ alors l'entropie associée H est le nombre réel < 1 défini par $H = -(f_1 \log f_1 + \ldots + f_n \log f_n)$. L'un des théorèmes principaux de Shannon assure que 1) tout codage en binaire de ce texte contient au moins nH bits, 2) il existe un codage en binaire avec $\leq n(H+1)$ bits.

Autour de 1964, Kolmogorov repense complètement la théorie de Shannon en terme de calculabilité. Au lieu de considérer les codages des lettres par des mots binaires, il regarde globalement le texte et considère la taille d'un plus court programme informatique (ou de machine) capable de l'écrire, c'est la complexité de Kolmogorov du texte. Par exemple, il y a un programme très court pour faire écrire $2^{1000000}$ (même si son temps d'exécution, lui, est de l'ordre du million) alors que pour faire écrire un nombre tiré au hasard avec un million de chiffres il faudra un programme de taille à peu près un million. Noter, en revanche, que l'écriture d'un million de digits du nombre π (le fameux $3,14\ldots$), se fait par un programme très court car on connaît des algorithmes simples pour calculer ces digits.
A priori, cette complexité doit dépendre fortement du langage de programmation (ou modèle de calculabilité). Kolmogorov montre qu'il n'en est rien : cette dépendance est bornee : étant donnés deux langages, il existe un entier d tel que les complexités de n'importe quel texte relativement à ces langages ne différent que d'au plus d. Comme le dit plaisamment Kolmogorov,

la complexité d'un texte comme “Guerre et Paix" peut être considérée comme bien définie indépendamment du langage, la variation étant négligeable.

Comme on peut s'y attendre, cette complexité, parfaitement définie sur le plan théorique, n'est cependant pas une fonction calculable... Ceci peut sembler rédhibitoire pour des applications mais il n'en est rien (cf. le livre (149)).
D'une part, la complexité de Kolmogorov permet de fonder la notion d'objet aléatoire, une notion que la théorie des probabilités (telle qu'axiomatisée par le même Kolmogorov en 1933) ignore, ne s'intéressant qu'aux ensembles d'objets (dont elle mesure la probabilité) et non pas aux objets eux-mêmes. C'est Per Martin-Löf, en 1965, alors élève de Kolmogorov, qui le premier formalise cette notion intuitive d'objet aléatoire. Depuis, d'autres formalisations ont été trouvées (Levin et Chaitin, indépendamment, vers 1973) qui sont toutes équivalentes.
D'autre part, Paul Vitanyi, vers 2000, en utilisant des approximations de la complexité de Kolmogorov à l'aide des compresseurs usuels (gzip,...), a créé une nouvelle méthode de classification des documents d'une très grande efficacité (43; 44; 148).

Remerciements. Olivier Bournez souhaite remercier Pablo Arrighi, Nachum Dershowitz et Yuri Gurevich pour leurs relectures, réactions et commentaires pertinents sur un document à propos de la thèse de Church qui a été partiellement utilisé pour la rédaction de certains paragraphes de ce chapitre

Références

[Aaronson et al.] Aaronson, S., Kuperberg, G., and Granade, C. Complexity zoo. `http ://qwiki.stanford.edu/wiki/Complexity_Zoo`.

[1] Abramsky, S., Gabbay, D. M., and Maibaum, T. S., editors (1992-2001). *Handbook of Logic in Computer Science*, volume 1–5. Elsevier.

[2] Adleman, L. M. (1974). Generalized first-order spectra and polynomial-time recognizable sets. complexity of computation. In *SIAM-AMS Proceedings*, volume 7, pages 43–73.

[3] Adleman, L. M. (1988). An abstract theory of computer viruses. In *Advances in Cryptology — CRYPTO'88*, volume 403. Lecture Notes in Computer Science.

[4] Ajtai, M. (1983). σ_1^1 formulae on finite structures. *Journal of Pure and Applied Logic*, 24 :1–48.

[5] Arora, S. and Barak, B. (2009). *Computational Complexity : A Modern Approach*. Cambridge University Press.

[6] Arrighi, P. and Dowek, G. (2008). A quantum extension of Gandy's theorem. Colloque "La thèse de Church : hier, aujourd'hui, demain".

[7] Asarin, E. and Collins, P. (2005). Noisy Turing machines. In *Proc. of ICALP'05*, volume 3580 of *Lecture Notes in Computer Science*, pages 1031–1042. Springer-Verlag.

[8] Avizienis, A. (1961). Signed-digit representations for fast parallel arithmetic. *IRE Transactions on electronic computers (now IEEE Transactions on electronic computers)*, 10(3) :389–400.

[9] Balcázar, J. L., Diaz, J., and Gabarró, J. (1990). *Structural Complexity*. EATCS Monographs on Theoretical Computer Science.

[10] Bar Hillel, Y., Gaifman, C., and Shamir, E. (1960). On categorial and phrase structure grammars. *Bulletin of the Research Council of Israel*, 9F.

[11] Beggs, E. J., Costa, J. F., Loff, B., and Tucker, J. V. (2008). Computational complexity with experiments as oracles. *Proceedings of the Royal Society of London-A*, 464(2098) :2777–2801.

[12] Bellantoni, S. and Cook, S. (1992). A new recursion-theoretic characterization of the poly-time functions. *Computational Complexity*, 2 :97–110.

[13] Benedikt, M. and Segoufin, L. (2009). Regular tree languages definable in fo and in fomod. *ACM Transactions on Computational Logic*, 11 :4 :1–4 :32.

[14] Berger, R. (1966). The undecidability of the domino problem. *Memoirs of the American Mathematical Society*, 66 :72 pages.

[15] Bernstein, E. and Vazirani, U. (1993). Quantum complexity theory. In *Proceedings of the 25th Annual ACM Symposium on the Theory of Computation*, pages 11–20, New York. ACM press.

[16] Berry, G. and Sethi, R. (1986). From regular expressions to deterministic automata. *Theoretical Computer Science*, 48 :117–126.

[17] Berstel, J. and Perrin, D., editors (1985). *Theory of codes*. Academic Press.

[18] Berstel, J. and Reutenauer, C. (1982). Recognizable formal power series on trees. *Theoretical Computer Science*, 18 :115–148.

[19] Blake, R. (1926). The paradox of temporal process. *The Journal of Philosophy*, 23(24) :645–654.

[20] Blass, A. and Gurevich, Y. (2003). Abstract state machines capture parallel algorithms. *ACM Transactions on Computation Logic*, 4(4) :578–651.

[21] Blass, A. and Gurevich, Y. (2008). Abstract state machines capture parallel algorithms : Correction and extension. *ACM Transactions on Computation Logic*, 9(3) :1–32.

[22] Blum, L., Cucker, F., Shub, M., and Smale, S. (1998). *Complexity and Real Computation*. Springer-Verlag.

[23] Blum, L., Shub, M., and Smale, S. (1989). On a theory of computation and complexity over the real numbers ; NP completeness, recursive functions and universal machines. *Bulletin of the American Mathematical Society*, 21(1) :1–46.

[24] Blum, M. (1967). A machine-independent theory of the complexity of recursive functions. *J. ACM*, 14(2) :322–336.

[25] Bojańczyk, M. and Colcombet, T. (2006). Tree-walking automata cannot be determinized. *Theoretical Computer Science*, 350 :164–173.

[26] Bojańczyk, M. and Colcombet, T. (2008). Tree-walking automata do not recognize all regular languages. *SIAM Journal on Computing*, 38 :658–701.

[27] Boker, U. and Dershowitz, N. (2008). The Church-Turing thesis over arbitrary domains. In Avron, A., Dershowitz, N., and Rabinovich, A., editors, *Pillars of Computer Science : Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday*, volume 4800, pages 199–229. Springer.

[28] Bonfante, G., Kaczmarek, M., and Marion, J.-Y. (2006). On abstract computer virology : from a recursion-theoretic perspective. *Journal of computer virology*, 1(3-4).

[29] Bournez, O. and Campagnolo, M. L. (2008). A survey on continuous time computations. In Cooper, S., Löwe, B., and Sorbi, A., editors, *New Computational Paradigms. Changing Conceptions of What is Computable*, pages 383–423. Springer-Verlag.

[30] Bournez, O., Campagnolo, M. L., Graça, D. S., and Hainry, E. (2006). The general purpose analog computer and computable analysis are two equivalent paradigms of analog computation. In Cai, J., Cooper, S. B., and Li, A., editors, *Theory and Applications of Models of Computation, Third International Conference, TAMC 2006, Beijing, China, May 15-20, 2006, Proceedings*, volume 3959 of *Lecture Notes in Computer Science*, pages 631–643. Springer.

[31] Bournez, O., Campagnolo, M. L., Graça, D. S., and Hainry, E. (2007). Polynomial differential equations compute all real computable functions on computable compact intervals. *Journal of Complexity*, 23(3) :317–335.

[32] Brent, R. (1976). Fast multiple-precision evaluation of elementary functions. *Journal of the Association for Computing Machinery*, 23 :242–251.

[33] Brüggemann-Klein, A. (1993). Regular expressions to finite automata. *Theoretical Computer Science*, 120(2) :197–213.

[34] Brüggemann-Klein, A. and Wood, D. (1998). One-unambiguous regular languages. *Information and Computation*, 142 :182–206.

[35] Brzozowski, J. A. and Leiss, E. (1980). On equations for regular languages, finite automata, and sequential networks. *Theoretical Computer Science*, 10 :19–35.

[36] Büchi, J. (1960a). On a decision method in a restricted second order arithmetic. In Press., S. U., editor, *Proc. Internat. Congr. on Logic, Methodology and Philosophy of Science*, pages 1–11.

[37] Büchi, J. (1960b). Weak second-order arithmetic and finite automata. *Z. Math. Logik Grundl. Math.*, 6 :66–92.

[38] Bush, V. (1931). The differential analyzer. A new machine for solving differential equations. *J. Franklin Inst.*, 212 :447–488.

[39] Buss, S. R. (1987). The Boolean formula value problem is in ALOGTIME. In ACM, editor, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York*, pages 123–131. ACM Press.

[40] Calude, C. and Pavlov, B. (2002). Coins, quantum measurements, and Turing's barrier. *Quantum Information Processing*, 1(1-2) :107–127.

[41] Chandra, A. K., Kozen, D. C., and Stockmeyer, L. J. (1981). Alternation. *Journal of the Association for Computing Machinery*, 28 :114–133.

[42] Church, A. (1936). An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58 :345–363. Reprinted in (63).

[43] Cilibrasi, R. and Vitanyi, P. (2005). Clustering by compression. *IEEE Trans. Information Theory*, 51(4) :1523–1545.

[44] Cilibrasi, R., Vitanyi, P., and de Wolf, R. (2004). Algorithmic clustering of music based on string compression. *Computer Music J.*, 28(4) :49–67.

[45] Cobham, A. (1965). The intrinsic computational difficulty of functions. In Bar-Hillel, Y., editor, *Proceedings of the International Conference on Logic, Methodology, and Philosophy of Science*, pages 24–30. North-Holland, Amsterdam.

[46] Codd, E. F. (1968). *Cellular automata*. Academic Press.

[47] Cohen, F. (1986). *Computer Viruses*. PhD thesis, University of Southern California.

[48] Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., and Tommasi, M. (2007). Tree automata techniques and applications. http ://tata.gforge.inria.fr/.

[49] Cook, S. A. (1971). The complexity of theorem proving procedures. *Proceedings Third Annual ACM Symposium on Theory of Computing*, pages 151–158.

[50] Cook, S. A. (1983). An overview of computational complexity. *Commun. ACM*, 26(6) :400–408.

[51] Cook, S. A. (2003). The importance of the P versus NP question. *JACM*, 50(1) :27–29.

[52] Cook, S. A. and R.A., R. (1973.). Time bounded random access machines. *Journal of Computer and System Sciences*, 7(4) :354–375.

[53] Copeland, B. J. (1997). The broad conception of computation. *American Behavioural Scientist*, 40 :690–716.

[54] Copeland, B. J. (Fall 2002). The Church-Turing thesis. In Zalta, E. N., editor, *The Stanford Encyclopedia of Philosophy*. Stanford University. Available online at : `http://plato.stanford.edu/entries/church-turing/`.

[55] Copeland, B. J. and Shagrir, O. (2007). Physical Computation : How General are Gandy's Principles for Mechanisms ? *Minds and Machines*, 17 :217–231.

[56] Copeland, B. J. and Sylvan, R. (January 1999). Beyond the universal turing machine. *Australasian Journal of Philosophy*, 77 :46–66.

[57] Coquand, T. and Huet, G. (1988). The Calculus of Constructions. *Information and Computation*, 76 :95–120.

[58] Courcelle, B. (2010). *Graph algebras and monadic second-order logic*. Cambridge University Press.

[59] Cousineau, D. and Dowek, G. (2007). Embedding Pure Type Systems in the lambda-Pi-calculus modulo. In Ronchi Della Rocca, S., editor, *Typed Lambda Calculi and Applications*, volume 4583, pages 102–117. Springer.

[60] Crabbé, M. (1974). Non-normalisation de ZF. Manuscript.

[61] Crabbé, M. (1991). Stratification and cut-elimination. *The Journal of Symbolic Logic*, 56(1) :213–226.

[62] Curien, P.-L. and Herbelin, H. (2000). The duality of computation. *SIGPLAN Notices*, 35(9) :233–243. International Conference on Functional Programming, ICFP'00, Montreal.

[63] Davis, M. (1965). *The Undecidable : Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions*. Raven Press. Reprinted by Dover Publications, Incorporated in 2004.

[64] Dejean, F. and Schützenberger, M. P. (1966). On a question from eggan. *Information and Control*, 9 :23–25.

[65] Delorme, M. (1999). An introduction to cellular automata. In Delorme, M. and Mazoyer, J., editors, *Cellular automata : a parallel model*, pages 3–51. Kluwer.

[66] Denis, F., Esposito, Y., and Habrard, A. (2006). Learning rational stochastic languages. In *19th Annual Conference on Learning Theory, COLT 2006*, volume 4005 of *Lecture Notes in Computer Science*, pages 274–288.

[67] Dershowitz, N. and Gurevich, Y. (2008). A natural axiomatization of computability and proof of church's thesis. *Bulletin. of Symbolic Logic*, 14(3) :299–350.

[68] Deutsch, D. (1985). Quantum theory, the Church-Turing principle and the universal quantum computer. *Proc. R. Soc. Lond. Ser. A*, A400 :97–117.

[69] Deutsch, D. (1989). Quantum computational networks. *Proc. R. Soc. Lond. A*, 425 :73.

[70] Deutsch, D. and Jozsa, R. (1992). Rapid solution of problems by quantum computation. *Proc. R. Soc. Lond. A*, 439 :553–558.

[71] Dexter, S., Boyle, P., and Gurevich, Y. (1997). Gurevich local evolving algebras and Schönage storage modification machines. *Journal of Universal Computer Science*, 3(4) :279–303.

[72] Dowek, G. (2007). *Les métamorphoses du calcul : une étonnante histoire de mathématiques*. Le Pommier.

[73] Dowek, G., Hardin, T., and Kirchner, C. (2003). Theorem proving modulo. *Journal of Automated Reasoning*, 31 :32–72.

[74] Dowek, G. and Werner, B. (2003). Proof normalization modulo. *The Journal of Symbolic Logic*, 68(4) :1289–1316.

[75] Droste, M., Kuich, W., and Vogler, H., editors (2009). *Handbook of Weighted Automata*. Springer Verlag.

[76] Dupont, P., Denis, F., and Esposito, Y. (2005). Links between probabilistic automata and hidden markov models : probability distributions, learning models and induction algorithms. *Pattern Recognition*, 38 :1349–1371.

[77] Durbin, R., Eddy, S., Krogh, A., and Mitchison, G. (1998). *Biological Sequence Analysis : Probabilistic Models of Proteins and Nucleic Acids.* Cambridge University Press.

[78] Eggan, L. C. (1963). Transition graphs and the star-height of regular events. *Michigan Math. Journal*, 10 :385–397.

[79] Ehrenfeucht, A. and Zeiger, P. (1976). Complexity measures for regular expressions. *Journal of Computer and System Sciences*, 12 :134–146.

[80] Ekman, J. (1994). *Normal proofs in set theory*. PhD thesis, Chalmers university of technology and University of Göteborg.

[81] Elgot, C. and Robinson, A. (1964). Random-Access Stored-Program Machines, an Approach to Programming Languages. *Journal of the Association for Computing Machinery*, 11(4) :365–399.

[82] Elgot, C. C. (1965). Decision problems of finite automata design and related arithmetics. *Transactions of American Math. Society*, 98 :21–52.

[83] Etessami, K. and Yannakakis, M. (2009). Recursive markov chains, stochastic grammars, and monotone systems of nonlinear equations. *Journal of the ACM*, 56(1) :1–65.

[84] Ferbus, M. and Grigorieff, S. (2009). ASM and operational algorithmic completeness of Lambda Calculus. *To appear*.

[85] Feynman, R. (1960). There's plenty of room at the bottom : an invitation to open up a new field of physics. *Engineering and Science (California Institute of Technology)*, 23(5) :22–36.

[86] Feynman, R. P. (1984). Quantum-mechanical computers. *J. Opt. Soc. Am. B*, 1 :464.

[87] Fürer, M. (2009). Faster integer multiplication. *SIAM Journal on Computing*, 39(3) :979–1005. Earlier version in Proc. 39th ACM STOC 2007 conf., pp. 57-66.

[88] Fürsr, M. L., Saxe, J. B., and Sipser, M. (1984). Parity, circuits, and the polynomial-time hierarchy. *Mathematical Systems Theory*, 17(1) :13–27.

[89] Furst, M. L., Saxe, J. B., and Sipser, M. (1984). Parity, circuits, and the polynomial-time hierarchy. *Mathematical Systems Theory*, 17(1) :13–27.

[90] Gandy, R. O. (1980). Church's thesis and principles for mechanisms. In Barwise, J., Keisler, H. J., and Kunen, K., editors, *The Kleene Symposium*, pages 123–148. North-Holland.

[91] Garey, M. R. and Johnson, D. S. (1979). *Computers and intractability. A guide to the theory of NP-completness*. W. H. Freeman.

[92] Gentzen, G. (1934). Untersuchungen über das logische schließen. *Mathematische Zeitschrift*, 39 :405–431.

[93] Gentzen, G. (1936). Die widerspruchfreiheit der reinen zahlentheorie. *Mathematische Annalen*, 112 :493–565.

[94] Girard, J.-Y. (1971). Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In 2*d Scandinavian Logic Symposium*, pages 63–92. North Holland.

[95] Girard, J.-Y. (1998). Light linear logic. *Inf. Comput.*, 143(2) :175–204.

[96] Glushkov, V. M. (1961). The abstract theory of automata. *Russian Mathematical Surveys*, 16 :1–53.

[97] Gödel, K. (1931). Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatschefte fur Mathematik und Physik*, 38 :173–198. Traduction anglaise dans (63).

[98] Gödel, K. (1958). Ü eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *Dialectica*, 12 :280–287.

[99] Goldreich, O. (2008). *Computational Complexity : A Conceptual Perspective*. Cambridge University Press.

[100] Graça, D. S. (2004). Some recent developments on shannon's general purpose analog computer. *Mathematical Logic Quarterly*, 50(4–5) :473–485.

[101] Graça, D. S. (2007). *Computability with Polynomial Differential Equations*. PhD thesis, Instituto Superior Técnico.

[102] Graça, D. S. and Costa, J. F. (2003). Analog computers and recursive functions over the reals. *Journal of Complexity*, 19(5) :644–664.

[103] Grädel, E. and Nowack, A. (2003). Quantum computing and abstract state machines. *Lecture Notes in Computer Science*, pages 309–323.

[104] Griffiths, T. V. (1968). The unsolvability of the equivalence problem for lambda-free nondeterministic generalized machines. *Journal of the ACM*, 15 :409–413.

[105] Grigorieff, S. (2006). Synchronization of a bounded degree graph of cellular automata with non uniform delays in time $D \log_m D$. *Theoretical Computer Science*, 356(1-2) :170–185.

[106] Grigorieff, S. and Valarcher, P. (2009). Local Evolving Algebras unify all kinds of sequential computation models. *STACS 2010, to appear*.

[107] Grover, L. K. (1996). A fast quantum mechanical algorithm for database search. In *Proceedings of the 28th annual ACM symposium on Theory of computing*, pages 212–219. ACM Press, New York.

[108] Grzegorczyk, A. (1957). On the definitions of computable real continuous functions. *Fundamenta Mathematicae*, 44 :61–71.

[109] Gurari, E. M. (1982). The equivalence problem for deterministic two-way sequential transducers is decidable. *SIAM Journal on Computing*, 11 :448–452.

[110] Gurevich, Y. (1985). A New Thesis, abstract 85T-68-203. *Notices of the American Mathematical Society*, 6 :317.

[111] Gurevich, Y. (1988a). Kolmogorov machines and related issues. *Bulletin EATCS*, 33 :71–82.

[112] Gurevich, Y. (1988b). Logic and the challenge of computer science. In Boerger, E., editor, *Current Trends in Theoretical Computer Science*, pages 1–50.

[113] Gurevich, Y. (2000). Sequential abstract state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1(1) :77–111.

[114] Hallnäs, L. (1983). *On normalization of proofs in set theory*. PhD thesis, University of Stockholm.

[115] Hamkins, J. D. (2002). Infinite time Turing machines. *Minds and Machines*, 12(4) :521–539.

[116] Hashiguchi, K. (1988). Algorithms for determining relative star height and star height. *Information and Computation*, 78 :124–169.

[117] Hillis, W. D. (1986). *The Connection Machine*. MIT Press (Partiellement disponible sur books.google.fr).

[118] Hillis, W. D. (1989). Richard Feynman and the Connection Machine. *Physics Today*, 42(2) :78–83.

[119] Hogarth, M. (1994). Non-Turing computers and non-Turing computability. *Philosophy of Science Association*, i :126–138.

[120] Hopcroft, J. and Ullman, J. (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

[121] Horner, W. G. (1819). A new method of solving numerical equations of all orders, by continuous approximation. *Philosophical Transactions of the Royal Society of London*, 2 :308–335.

[122] Immerman, N. (1986). Relational queries computable in polynomial time. *Information and Control*, 68(1-3) :86–104.

[123] Jain, S., Osherson, D., Royer, J. S., and Sharma, A. (1999). *Systems that learn*. MIT press.

[124] Jiang, T. (1992). The synchronization of non-uniform networks of finite automata. *Information and Computation*, 97(2) :234–261.

[125] Johnson, D. S. (1990). A catalog of complexity classes. In *(228)*, volume A, pages 67–161. Elsevier.

[126] Jones, J. P. (1982). Universal diophantine equation. *J. Symbolic Logic*, 47 :549–571.

[127] Jones, N. D. (1997). *Computability and complexity, from a programming perspective*. MIT press.

[128] Jones, N. D. and Selman, A. L. (1974). Turing machines and the spectra of first-order formulas. *J. Symb. Log.*, 39(1) :139–150.

[129] Karatsuba, A. A. and Ofman, Y. (1962). Multiplication of many-digital numbers by automatic computers. *Doklady Akad. Nauk SSSR*, 145 :293–294.

[130] Karp and Ramachandran (1990). Parallel algorithms for shared memory machines. In Leeuwen, J. V., editor, *Handbook of Theoretical Computer Science A : Algorithms and Complexity*, pages 870–941. Elsevier Science Publishers and The MIT Press.

[131] Kleene, S. (1956). *Representation of Events in Nerve Nets and Finite Automata*, pages 3–42. Princeton University Press.

[132] Kleene, S. C. (1936). General recursive functions of natural numbers. *Mathematical Annals*, 112 :727–742. Reprinted in (63).

[133] Knuth, D., Morris, J., and Pratt, V. (1977). Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2) :323–350.

[134] Knuth, D. E. (1973). *The Art of Computer Programming*, volume 3 : Sorting and Searching. Addison-Wesley.

[135] Knuth, D. E. (1981). *The Art of Computer Programming*, volume 2 : Seminumerical algorithms. Addison-Wesley. 2d edition.

[136] Kolmogorov, A. N. (1953). On the notion of algorithm. *Uspekhi Mat. Naut*, 8 :175–176. In russian.

[137] Kolmogorov, A. N. and Uspensky, V. (1958). On the definition of an algorithm. *Uspekhi Mat. Naut*, 13(4). English translation in AMS translation vol. 21 (1963), 217-245.

[138] Krivine, J.-L. and Parigot, M. (1990). Programming with proofs. *J. Inf. Process. Cybern. EIK*, 26(3) :149–167.

[139] Lacombe, D. (1955). Extension de la notion de fonction récursive aux fonctions d'une ou plusieurs variables réelles III. *Comptes Rendus de l'Académie des Sciences Paris*, 241 :151–153.

[140] Lacombe, D. (1960). La théorie des fonctions récursives et ses applications. *Bulletin de la Société Mathématique de France*, 88 :393–468.

[141] Leeuw, K. d., Moore, E. F., Shannon, C. E., and Shapiro, N. (1956). Computability by probabilistic machines. In Shannon, C. and MacCarthy, J., editors, *Automata Studies*, pages 183–212. Princeton University Press.

[142] Leivant, D. (1983). Reasoning about functional programs and complexity classes associated with type disciplines. In *Foundations of Computer Science, FOCS'83*, volume 88, pages 460–469.

[143] Leivant, D. (1991). A foundational delineation of computational feasiblity. In *Proceedings of the Sixth IEEE Symposium on Logic in Computer Science (LICS'91)*.

[144] Leivant, D. (1994). Predicative recurrence and computational complexity I : Word recurrence and poly-time. In Clote, P. and Remmel, J., editors, *Feasible Mathematics II*, pages 320–343. Birkhäuser.

[145] Leivant, D. and Marion, J.-Y. (1993). Lambda calculus characterizations of poly-time. *Fundamenta Informaticae*, 19(1,2) :167,184.

[146] Levin, L. (1973). Universal search problems. *Problems of information transmission*, 9(3) :265–266.

[147] Levin, L. (1986). Average case complete problems. *SIAM Journal on Computing*, 15(1) :285–286.

[148] Li, M., Chen, X., Li, X., Ma, B., and Vitanyi, P. (2004). The similarity metric. *IEEE Trans. Inform. Th*, 50(12) :3250–3264.

[149] Li, M. and Vitanyi, P. (2008). *An Introduction to Kolmogorov Complexity and its Applications*. Springer-Verlag. 3rd edition.

[150] Libkin, L. (2004). *Elements of Finite Model Theory*. Springer.

[151] Longo, G. and Paul, T. (2009). Le monde et le calcul. Réflexions sur calculabilité, mathématiques et physique. In *Logique et Interaction : Géométrie de la cognition*, volume Actes du colloque et école thématique du CNRS "Logique, Sciences, Philosophie", Cerisy. Hermann.

[152] Macintyre, A. J. and Wilkie, A. (1995). On the decidability of the real exponential field. In Odifreddi, P. G., editor, *Kreisel 70th Birthday Volume*. CLSI.

[153] Manning, C. and Schütze, H. (1999). *Foundations of Statistical Natural Language Processing*. The MIT Press, Cambridge.

[154] Margenstern, M. (2007). The domino problem of the hyperbolic plane is undecidable. *Bulletin of the EATCS*, 93 :220–237.

[155] Margenstern, M. (2008). The domino problem of the hyperbolic plane is undecidable. *Theoretical Computer Science*, 407 :28–84.

[156] Marion, J.-Y. (2001). Actual arithmetic and feasibility. In *CSL*, volume 2142 of *Lecture Notes in Computer Science*, pages 115–129.

[157] Martin-Löf, P. (1984). *Intuitionistic type theory*. Bibliopolis.

[158] Matijasevich, Y. (1977). Some purely mathematical results inspired by mathematical logic. In *Proceedings of Fifth International Congress on Logic, Methodology and Philosophy of science, London, Ontario, 1975*, pages 121–127. Dordrecht : Reidel.

[159] Mayr, E. W. (1984). An algorithm for the general Petri net reachability problem. *SIAM Journal on Computing*, 13(3) :441–460. Preliminary version in Proceedings of the thirteenth annual ACM symposium on Theory of computing, p. 238–246, 1981.

[160] Mazoyer, J. and Yunès, J.-B. (2010). Zzz. In XX, C. and ZZ, J., editors, *XXX*, pages 00–00. AAA.

[161] McNaughton, R. (1966). Testing and generating infinite sequences by a finite automaton. *Information and Control*, 9 :521–530.

[162] McNaughton, R. and Papert, S. (1971). *Counter-Free Automata*. The MIT Press.

[163] McNaughton, R. and Yamada, H. (1960). Regular expressions and state graphs for automata. *Transactions Electronic Computers*, 9 :39–47.

[164] Melzak, Z. (1961). An informal arthmetical approach to computability and computation. *Canadian Mathematical Bulletin*, 4(3) :279–293.

[165] Minsky, M. (1961). Recursive unsolvability of post's problem of 'tag' and other topics in theory of turing machines. *Annals of Mathematics*, 74 :437–455.

[166] Moschovakis, Y. (2001). What is an algorithm ? In Engquist, B. and Schmid, W., editors, *Mathematics unlimited – 2001 and beyond*, pages 919–936. Springer.

[167] Muller, J.-M. (1989). *Arithmétique des ordinateurs*. Masson.

[168] Negri, S. and von Plato, J. (2001). *Structural proof theory*. Cambridge University Press.

[169] Nielsen, M. A. (1997). Computable functions, quantum measurements, and quantum dynamics. *Physical Review Letters*, 79(15) :2915–2918.

[170] Ord, T. (2002). Hypercomputation : computing more than the turing machine. *Arxiv preprint math/0209332*.

[171] Ord, T. (2006). The many forms of hypercomputation. *Applied Mathematics and Computation*, 178(1) :143–153.

[172] Ord, T. and Kieu, T. D. (2004). Using biased coins as oracles. *Arxiv preprint cs/0401019*.

[173] Pan, V. Y. (1966). Methods of computing the values of polynomials. *Russian Mathematical Surveys*, 21(1) :105–137.

[174] Papadimitriou, C. H. (1994). *Computational Complexity*. Addison-Wesley.

[175] Parigot, M. (1992). $\lambda\mu$-calculus : An algorithmic interpretation of classical natural deduction. In Voronkov, A., editor, *Logic Programming and Automated Reasoning*, volume 624, pages 190–201. Springer.

[176] Paul, W. (1979). Kolmogorov complexity and lower bounds. In Budach, L., editor, *Second Int. Conf. on Fundamentals of Computation Theory*, pages 325–334. Akademie Berlin.

[177] Paul, W., Pippenger, N. J., Szemerdi, E., and Trotter, W. T. (1983). On determinism versus nondeterminism and related problems. In *Proceedings of IEEE FOCS'83*, pages 429–438.

[178] Paulin-Mohring, C. (1993). Inductive definitions in the system Coq - rules and properties. In Bezem, M. and de Groote J.F., editors, *Typed lambda calculi and applications*, volume 664, pages 328–345. Springer.

[179] Paulin-Mohring, C. and Werner, B. (1993). Synthesis of ml programs in the system coq. *J. Symb. Comput.*, 15(5/6) :607–640.

[180] Paz, A. (1971). *Introduction to Probabilistic Automata*. Academic Press.

[181] Pentus, M. (1997). Product-free Lambek calculus and context-free grammars. *Journal of Symbolic Logic*, 62(2) :648–660.

[182] Plandowski, W. (1995). *The complexity of the morphism equivalence problem for context-free languages*. PhD thesis, Warsaw University. Department of Informatics, Mathematics, and Mechanics.

[183] Poibeau, T. (2003). *Extraction automatique d'information. Du texte brut au web sémantique*. Hermès.

[184] Prawitz, D. (1965). *Natural deduction, a proof-theoretical study*. Almqvist & Wiksell.

[185] Rabin, M. (1969). Decidability of Second-Order Theories and Automata on Infinite Trees. *Transactions of the American Mathematical Society*, 141 :1–35.

[186] Rabin, M. O. (1987). *ACM Turing Award Lecture, 1977 : Complexity of computations*, pages 625–633. ACM Press/Addison-Wesley Publishing Co.

[187] Robinson, J. (1949). Definability and decision problems in arithmetic. *Journal of Symbolic Logic*, 14 :98–114.

[188] Rosenstiehl, P. (1986). Existence d'automates finis capables de s'accorder bien qu'arbitrairement connectés et nombreux. *Internat. Comp. Centre*, 5 :245–261.

[189] Rozenberg, G. and Salomaa, A., editors (1997). *Handbooks of Formal Languages : vol 1. Word, language, grammar ; vol 2. Linear modeling ; vol 3. Beyond words*. Springer Verlag.

[190] Rubin, S. (2008). Automata presenting structures : A survey of the finite string case. *Bulletin of Symbolic Logic*, 14(2) :169–209.

[191] Russell, B. (1935). The limits of empiricism. In *Proceedings of the Aristotelian Society*, volume 36, pages 131–150. Blackwell Publishing ; The Aristotelian Society.

[192] Sakarovitch, J. (2009). *Elements of Automata Theory*. Cambridge University Press.

[193] Savage, J. (1998). *Models of computation, Exploring the power of computing*. Addison Wesley.

[194] Savitch, W. J. (1970). Relationship between nondeterministic and deterministic tape classes. *JCSS*, 4 :177–192.

[195] Schönhage, A. (1980). Storage modification machines. *SIAM Journal on Computing*, 9(3) :490–508.

[196] Schönhage, A. and Strassen, V. (1971). Schnelle multiplikation großer zahlen. *Computing*, 7 :281–292.

[197] Sch "utte, K. (1951). Beweisetheoretische erfassung der unend lichen induktion in der zahlentheorie. *Mathematische Annalen*, 122 :369–389.

[198] Schützenberger, M. P. (1965). On finite monoids having only trivial subgroups. *Information and Control*, 8 :190–194.

[199] Seiferas, J. I. (1990). Machine-independent complexity. In Leeuwen, J. V., editor, *Handbook of Theoretical Computer Science A : Algorithms and Complexity*, pages 163–186. Elsevier Science Publishers and The MIT Press.

[200] Sénizergues, G. (1998). The equivalence problem for deterministic pushdown transducers into abelian groups. In *MFCS*, pages 305–315.

[201] Sénizergues, G. (2002). L(A)=L(B) ? a simplified decidability proof. *Theoretical Computer Science*, 281 :555–608.

[202] Shannon, C. E. (1941). Mathematical theory of the differential analyser. *Journal of Mathematics and Physics MIT*, 20 :337–354.

[203] Shannon, C. E. (1948). The mathematical theory of communication. *Bell System Tech. J.*, 27 :379–423.

[204] Shepherdson, J. C. and Sturgis, H. E. (1963). Computability of recursive functions. *J. ACM*, 10(2) :217–255.

[205] Shor, P. (1994). Algorithms for quantum computation : Discrete logarithms and factoring. In Shafi Goldwasser, I. C. S. P., editor, *Proceedings of the 35nd Annual Symposium on Foundations of Computer Science*, pages 124–134.

[206] Sieg, W. (1994). Mechanical procedures and mathematical experience. *Mathematics and mind*, pages 71–117.

[207] Sieg, W. (1997). Step by recursive step : Church's analysis of effective calculability. *The Bulletin of Symbolic Logic*, 3(2) :154–180.

[208] Sieg, W. (1999). Hilbert's programs : 1917-1922. *The Bulletin of Symbolic Logic*, 5(1) :1–44.

[209] Sieg, W. (2008). Church without dogma–Axioms for computability. In Cooper, S., Löwe, B., and Sorbi, A., editors, *New Computational Paradigms. Changing Conceptions of What is Computable*, New York. Springer.

[210] Siegelmann, H. T. and Fishman, S. (1998). Analog computation with dynamical systems. *Physica D*, 120 :214–235.

[211] Siegelmann, H. T. and Sontag, E. D. (1995). On the computational power of neural nets. *Journal of Computer and System Sciences*, 50(1) :132–150.

[212] Simon, D. (1994). On the power of quantum computation. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 116–123.

[213] Sipser, M. (2006). *Introduction to the Theory of Computation (2nd ed.)*. Thomson Course Technology.

[214] Slot, C. F. and van Emde Boas, P. (1984). On tape versus core an application of space efficient perfect hash functions to the invariance of space. In ACM, editor, *Proceedings of the sixteenth annual ACM Symposium on Theory of Computing, Washington, DC, April 30–May 2, 1984*, pages 391–400. ACM Press.

[215] Smith, W. D. (1999). History of "Church's theses" and a manifesto on converting physics into a rigorous algorithmic discipline. Technical report, NEC Research Institute. Avalaible on `http://www.math.temple.edu/~wds/homepage/works.html`.

[216] Smullyan, R. M. (1961). *Theory of Formal Systems*. Princeton University Press.

[217] Smullyan, R. M. (1993). *Recursion Theory for Metamathematics*. Oxford University Press.

[218] Smullyan, R. M. (1994). *Diagonalization and Self-Reference*. Oxford University Press.

[219] Smullyan, R. M. (1995). *Théorie de la récursivité pour la métamathématique*. Masson, Paris. Traduction Philippe Ithier.

[220] Spaan, E., Torenvliet, L., and van Emde Boas, P. (1989). Nondeterminism fairness and a fundamental analogy. *Bulletin of the EATCS*, 37 :186–193.

[221] Tait, W. W. (1967). Intentional interpretations of functionals of finite type I. *The Journal of Symbolic Logic*, 32 :198–212.

[222] Tarski, A. (1931). Sur les ensembles définissables de nombres réels I. *Fundamenta Mathematica*, 17 :210–239.

[223] Thatcher, J. W. and Wright, J. B. (1968). Generalized finite automata with an application to a decision problem of second-order logic. *Mathematical System Theory*, 2 :57–82.

[224] Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2) :230–265. Reprinted in (63).

[225] Turing, A. M. (1939). Systems of logic based on ordinals. *Proceedings of the London Mathematical Society*, 45 :161–228.

[226] Urban, C. (2001). Strong Normalisation for a Gentzen-like Cut-Elimination Procedure. In *Typed lambda calculi and applications*, volume 2044, pages 415–429. Springer.

[227] van Emde Boas, P. (1990). Machine models and simulations. In *Handbook of theoretical computer science (vol. A) : Algorithms and Complexity*.

[228] van Leeuwen, J., editor (1990a). *Handbook of theoretical computer science*, volume A : Algorithms and Complexity, B : Formal Models and Semantics. The MIT Press and Elsevier.

[229] van Leeuwen, J., editor (1990b). *Handbook of theoretical computer science*, volume A : Algorithms and Complexity, B : Formal Models and Semantics. The MIT Press and Elsevier.

[230] Vergis, A., Steiglitz, K., and Dickinson, B. (1986). The complexity of analog computation. *Mathematics and Computers in Simulation*, 28(2) :91–113.

[231] von Neumann, J. (1951). A general and logical theory of automata. In Jeffries, L., editor, *Cerebral Mechanisms in Behavior–The Hixon Symposium*, pages 1–31. Wiley. Reprinted in *Papers of John von Neumann on Computing and Computer Theory*, ed. William Aspray and Arthur Burks, MIT Press, 1987.

[232] von Neumann, J. (1966). *Theory of Self-Reproducing Automata*. University of Illinois Press. (Edité et complété par Arthur W. Burks à partir d'un cours donné par von Neumann en 1949).

[233] Wack, B. (2005). *Typage et déduction dans le calcul de réécriture*. PhD thesis, Université Henri Poincaré, Nancy 1.

[234] Weihrauch, K. (2000). *Computable Analysis : an Introduction*. Springer.

[235] Werner, B. (1994). *Une théorie des constructions inductives*. PhD thesis, Université Paris 7.

[236] Weyl, H. and Kirschmer, G. (1927). *Philosophie der Mathematik und Naturwissenschaft*. Oldenbourg Wissenschaftsverlag.

[237] Wolfram, S. (2002). *A New Kind of Science*. Wolfram Media.

[238] Yao, A. (1993). Quantum circuit complexity. In *Proc. 34th IEEE Symposium on Foundation of Computer Science*.

[239] Yao, A. C.-C. (2003). Classical physics and the Church–Turing Thesis. *Journal of the ACM*, 50(1) :100–105.

[240] Yunès, J.-B. (2006). Fault tolerant solutions to the firing squad synchronization problem in linear cellular automata. *Journal of Cellular Automata*, 1(3) :253–268.